Exp[redacted] Mail Label No.:
EL [redacted]712 771 US
Date of Deposit: November 9, 1998

Broadcom Corporation
Provisional Patent Application

Title: **GRAPHICS CHIP ARCHITECTURE**

Inventors: Alexander G. MacInnis,
San Carlos, California,
Citizen of USA

Chungfuh Jeffrey Tang,
Saratoga, CA;
Citizen of USA

Xiaodong Don Xie,
Santa Clara, CA;
Citizen of People's Republic of China

James Patterson
Santa Clara, CA
Citizen of USA

Greg Kranawetter,
San Jose CA;
Citizen of USA

1 Background of the Invention.

This document describes the architecture of a display engine, referred to herein as the Sapphire graphics engine, which is tailored for use in advanced versions of television control electronics, such as set top boxes, integrated digital TVs, and home network computers.

The display engine is that part of the graphics engine which receives display pixel data from any combination of locally attached memory and digital video input ports, processes these in some way, and produces final display pixels as output. It does not include the acceleration functions which read and write main graphics memory, in the way that a processor or a blitter would.

The Sapphire display engine design was produced from a top-down, first-principles look at the requirements of the applications that are expected to run on the target products (e.g. STBs). While there is currently no source for a fixed set of requirements, requirements were derived from reasonable expectations about applications, services, tools, formats, and protocols.

The architecture specified here is grouped into separate sections for "graphics" and "video". There is a technical reason for this, based on the same analysis that produced this architecture, and it reflects the structure of the hardware itself. This split does not, however, imply the existence of any fundamental difference between graphics and video, and in fact much of the functionality is common to both.

Brief Description of the Figures.

Fig. 1 is a block diagram of a complete graphics and video display pipeline according to the present invention.

Fig. 2 is block diagram of a portion of the Graphics display pipeline of Fig. 1.

Fig. 3 shows a Four-tap Gaussian filter kernel.

Fig. 4 shows a Two-tap bilinear filter kernel.

Fig. 5 is a window controller block diagram for window list sorting.

Fig. 6 is a DMA block diagram is provided for use with the present invention.

Fig. 7 is a video_sub component of the present invention.

Fig. 8 is a VDIN block diagram for a time base corrector used in the present invention.

Fig. 9 is a video scalar block according to the present invention.

Fig. 10 is the graphics display engine of the present invention.

Fig. 11 is a state diagram of the state machine of the present invention.

Fig. 12 shows the data flow path of a graphics converter according to the present invention.

Fig. 13 shows the data flow path for conversion and blending in the present invention.

Fig. 14 shows the structure of the graphics color look up table (CLUT) of the present invention.

Fig. 15 is a graphical representation if the write port timing according to the present invention.

Fig. 16 shows a graphics line buffer according to the present invention.

Fig. 17 is a block diagram of the digital video decoder according to the present invention.

Fig. 18 is a table is provided for the port definition of the present invention.

Summary of the Invention.

The display engine of the present invention is for use on normal TVs, in both US and foreign markets. It produces very high quality display of text and graphics on normal TVs, along with video from various sources. Graphics content may be produced by software optimized for image quality - and from other software which may treat the device as if it were a dumb frame buffer on a PC. Video content may be scaled up or down by a factor covering a continuum from much less than 1 up to 4. In such cases the video data will come from memory, either the Sapphire local memory or that of another closely connected device. Video data may also come from an external data stream source, such as an MPEG decoder or an analog composite video decoder. This may be referred to as "passthrough" video.

Graphics images may appear in separate logical planes, and in separate logical windows (or regions, or viewports). These windows may overlap, and they may be blended together

in various ways. There may also be a pointer (sometimes referred to as a cursor). There are a few basic reasons why graphics windows should preferably be supported in hardware in the Sapphire engine:

- Save memory: There are cases, such as vertical bars of graphics on the left and right, but not in the middle, where it would be wasteful to allocate memory for the region in the middle where there is no graphics. With two regions or more per line, this memory can be saved. Memory is also saved when animations are done by changing display window descriptors, rather than by saving, writing, and restoring graphics images in a single display memory. If a single buffer per window can be used, this creates a very significant memory savings when compared with the method typical of PCs, which is to double buffer the display and "flip" buffers whenever a new buffer is ready to display.

- Dissolves: With two planes that can completely overlap each other and be blended with each other using an arbitrary blend factor, it becomes trivial to dissolve the two layers, without needing to read, modify and write the frame buffer contents for each new blend factor, nor is it necessary to store the result of the blend in memory.

- Blending: Apart from dissolves, it is often a requirement to blend various graphics and video images together in arbitrary ways. With the video content being updated every display field, this blending would generally not be practical in software, while it is readily handled in a display engine designed with a powerful windowing capability.

- Pointers: A pointer is essentially a small window. Hardware support for a pointer simplifies the requirements for software considerably, as without hardware, it would be necessary to save, write, and restore graphics images every time the pointer is moved or changed and every time the imagery behind it changes. Pointers may have somewhat less requirements than other graphics windows, but fundamentally they have similar requirements.

- Mixing graphics of different types: Various graphics images may have images in CLUT, RGB or YUV colors

spaces. With separate windows each can be handled in its native format, while if one common buffer were used, one or more images would have to be converted to a common format which reflects the current display mode of the hardware. Generally this would mean RGB16 (or more bits), since it can support almost all colors which are supportable by CLUT, but this can take memory and bandwidth, and it can cause the loss of some functions.

- Alpha per pixel: Related to the point about mixing different types of graphics, some formats may have an alpha-per-pixel capability, while others may not. Typical examples include alphaCLUT8, which has an alpha value per CLUT entry, and RGB16, which has no alpha per pixel capability. Converting CLUT to RGB would lose the alpha per color (or per pixel) capability.

- Animation and scrolling: A key requirement of text-based applications is smooth soft scrolling, and also animated screen objects often need to be animated by moving around on the screen, changing the look of the object, or both. If an object is a window, it can easily change both its position and its appearance simply by changing descriptor values and/or the contents of that one window, without regard to the rest of the display. Scrolled text is an obvious and powerful capability, and often only one such window is required to be animated at a time (there may be multiple text windows, and the ones that are not moving do not need to be their own windows.) Animated graphics can create an unbounded requirement, since it is very difficult to put a limit on the number of such objects that may appear on the screen, or on any one line of the screen, at once. Nevertheless, it is a useful function, and applications can take advantage of it if the application creators understand the capabilities of the display engine. Note that if there are many animated sprites on screen, it may be better to allocate one memory plane to display the entire area where the sprites can exist on the screen, and animate sprites within that memory using an acceleration engine, which may exist but is not covered here.

- Scaling: It may be necessary to scale some windows on the screen separately from others. For example, some graphics content may be designed for square aspect ratio pixel display, requiring scaling in either the horizontal or vertical axes, while others may be designed for display on TVs with 720 pixels per line (and any combination of 625/50 and 525/60 scan rates and 4:3 or 16:9 display aspect ratios.) Video content, on the other hand, may require scaling up or down in a symmetrical horizontal and vertical fashion, with arbitrary scale factors.

Detailed Description of a Preferred Embodiment(s)

1.1 Graphics window display requirements

Specific requirements for the Sapphire display engine to display the graphics windows, then, include the following:

1. Three graphics windows, including one intended primarily for the pointer, can be displayed on any one scan line. There may be more windows on different display lines.

2. The two main graphics windows can have any format from the following list: CLUT2, CLUT4, CLUT8 (all with alpha per color), RGB15+1, RGB16, and YUV 4:2:2. Since graphics windows can have a YUV 4:2:2 color space, they can display live video if video is being written to memory in real time, however it is not a requirement to perform arbitrary scaling of YUV graphics windows. There are scaled YUV video windows described in the section on video window requirements.

3. The pointer window is defined to be on top of all other windows. While alpha blending is not an absolute requirement if provides a nice touch and it is easily provided by the architecture specified here.

4. There is one CLUT active at a time, and it is shared by all graphics windows that use CLUT formats. The CLUT can be re-loaded automatically when it is not being used. Since this could take tens of microseconds, however, the application must leave at least one display line in which there are no CLUT format windows displayed in order to allow enough time to load the CLUT from memory.

5. Graphics windows have chroma key capability, which specifies transparency as a CLUT pixel matching a reference value or an RGB pixel being within a min-max range. RGB15+1 uses the 1 keying bit instead of chroma key. Note that alpha per entry with the CLUT is a superset of the CLUT chroma key, so special logic is needed only for RGB16 chroma key.

6. Alpha for each pixel can optionally be derived from the Y (luma) value created when pixels (particularly RGB) are converted to YUV. In the simplest implementation, Y is simply used as the alpha value. In more powerful versions, the alpha value is looked up in a table, using Y as the index.

7. Each graphics window can have its own constant alpha value, instead of using the alpha per color or alpha from Y. Transparency keying is still supported when using this function.

8. Specification of the alpha blending values for graphics is defined to blend each graphics layer with the composition of all layers behind it, even if those layers are video windows. The implementation of blending in the engine is not, however, the obvious one of building up all layers from back to front; yet, the alpha values operate as if the implementation were this obvious one. The implementation is explained in later sections.

9. There is an anti-flicker (or anti-flutter) filter, called AFF, which can optionally filter all the graphics windows (including the pointer) as a group to eliminate flutter on interlaced TVs. This filter addresses the problem at the edges of the graphics windows as well as within them.

10. There is a sample rate converter (SRC) which optionally scales all the graphics windows as a group to correct the aspect ratio, allowing square pixel content to appear with the correct aspect ratio on TVs with 720 pixels per line, with either 625/50 or 525/60 scan rates. This SRC is designed for 4:3 displays, not 16:9 displays, but may be modified for square pixels on 16:9 displays.. The SRC works by scaling the graphics up or down by a small amount (about 11%) in the vertical dimension. There is no horizontal SRC for graphics.

11. Visually, all the graphics windows appear on top of all video windows and the background color.

1.2 Video window display requirements

Video windows have requirements with some similarities to and some differences from those of graphics. These include:

1. One scaled video window per display line can be displayed from memory. This window has the capability of scaling the video to an arbitrary scale factor ranging from much less than 1 up to 4. When scaling to less than 1, the scaling is performed in the capture engine, and when scaling is to greater than 1, the scaling is performed in the display engine.

2. One video window can be displayed from a passthrough source, such as MPEG or decoded analog video; this is referred to as the passthrough window.

3. The scaled video window is in either YUV 4:2:2 or YUV 4:2:0 format. In this case 4:2:0 is defined as field based, that is, the UV pair is sub-sampled from within the same field, not the adjacent line in the other field, and may be added later.

4. The scaled video window supports its own constant alpha value, for blending with the layers behind it.

5. The scaled video window can be scaled up in both horizontal and vertical dimensions by the display engine. The scale factor used in the display engine can range from 1.0 to 4.0. The scaling function uses a four-tap vertical filter and an 8 tap horizontal filter. All coefficients are programmable. The phase accuracy of the SRC is at least as fine as one-fourth of a pixel in each dimension; we may want to support one-eighth pixel accuracy in the horizontal dimension. It may be necessary to require one or more display scan lines between vertically adjacent scaled video windows, with no scaled video content in these lines, in order to allow time to process all the data in the vertical axis video scaler.

6. The video display scaling engine can be used for video down scaling in the input data path, although not at the same time as it is being used for display scaling. Such re-purposing of this engine may require switching clock domains, depending on other aspects of the device implementation.

7. Behind the scaled video window can be a passthrough video window. The passthrough window cannot

be blended with the background color, and it does not have any keying capability. Its size and location are defined by a rectangle.

8. Behind the passthrough video window can be a solid background color. This covers the entire screen, i.e. all that is not blanked.

2 Architecture Overview

The Sapphire engine display pipeline architecture consists of two major sections, the graphics display pipeline and the video display pipeline. These are shown in Fig. 1.

The following sections describe each of these pipelines briefly, and then later sections explain each block in detail. The blocks in the diagram above are logical blocks and do not necessariy represent exactly the implementation; for example the FIFOs are logically separate but are likely to be implemented in a shared SRAM in order to be more silicon-efficient.

A drawing of the video display pipeline will show the data paths for using the scaler to down-scale video from an input and writing the result to memory, in addition to the path shown above which is for up-scaling the video after reading it from memory.

2.1 Graphics Display Architecture

We have carefully considered and analyzed a variety of possible hardware architectures in order to find the best one that meets these requirements. Some of the difficulties in finding a good solution have to do with the need to perform scaling, anti-flutter filtering, and alpha blending of graphics at the same time, without creating a particularly complex design, while also minimizing the amount of on-chip memory required as well as minimizing the size and bandwidth required from off-chip memory..

The graphics display pipeline is shown schematically in Fig. 2.

2.1.1 DRAM

The memory block on the left refers to the bulk DRAM which is assumed to be attached to the Sapphire engine. The right arrow at the right edge of the diagram is the output from the graphics portion of the display engine. This output serves as the input to a subsequent block which blends graphics with video.

2.1.2 Window control

The window control block gets and stores window descriptors from main DRAM, and uses these descriptors to control the operation of the other blocks in the display pipeline. At any one time it stores, at a minimum, descriptors for all of the windows which are active on the current scan line. Descriptors include pertinent information such as the starting and ending line and pixel on the screen, the starting address in memory, the memory pitch, the pixel map format, alpha for the window, the depth number ("Z") of the window, and other parameters. On each scan line, one window is processed at a time, covering the extent from the left edge to the right edge of the screen, before proceeding to the next window. Windows are processed in order from back to front. The maximum number of windows that can be processed is limited, in this architecture, primarily by the performance of the pipeline - i.e. how many windows can be processed in the time allotted, and by the number of window descriptors the window control block can store at one time. Both of these parameters scale readily with increasing silicon density and speed.

2.1.3 DMA

The DMA block retrieves data from memory as needed to construct the various graphics windows. The addressing information for each graphics window is provided by a window control block which is not shown. Once the display of a window has begun, the DMA engine retains the key parameters needed to continue to read the required data from memory, including the current read address, the address of the start of the next line, the number of bytes to read per line, and the pitch.

The DMA operation reads all the contents of one line of one window of graphics at a time, starting with the back-most

graphics window which is visible on the current line. It does not read data from all the windows in parallel. This fact can is used to simplify the design of the DMA and all succeeding blocks in the display pipeline.

Since the display pipeline includes a vertical filter block for anti-flutter and scaling purposes, this block needs access to a set of adjacent lines, including those from both fields of the interlaced final display. Therefore all lines, not just those from the current display field, must be read from memory and processed during every display field. The effective rate of reading and processing graphics is equivalent to that of a non-interlaced display with a frame rate equal to the field rate of the interlaced display.

2.1.4 FIFO

The FIFO provides temporary storage for data which are read from memory, and it provides this data on demand to the converter block. The FIFO may also serve to bridge a boundary between different clock domains, in the event that in the implementation the memory and DMA use a different clock frequency or phase from the conversion and blending blocks. It is possible, but unlikely, that in some implementations the FIFO block may not be needed, that is, if the converter block can process data from memory at the rate that it is read from memory, and if the memory and conversion functions are in the same clock domain.

2.1.5 Converter

The converter block takes raw graphics data from the FIFO and converts it into a common YUValpha (YUVa) format. This YUVa format includes YUV 4:2:2 plus an 8-bit alpha value for every pixel, and as such it occupies 24 bits per pixel. [YUV is really a misnomer, its proper name is YcbCr, but YUV is easier to say and type, and it is in common use.] To convert from the raw graphics format to YUVa may require the use of the CLUT, in the case of CLUT formats, or it may require the use of a color space converter, in the case of RGB formats. The alpha value which is included in the YUVa output depends on a number of parameters, including:

- Alpha from chroma keying (transparent => alpha = 0)

- Alpha per CLUT entry
- Alpha from Y
- Alpha per window

The converter block receives information about the attributes of the window currently being converted from the window control block. As noted above under the DMA block, an entire line of one graphics window is processed before going on to the next graphics window, and the order in which they are processed is back to front. So, the attributes that the converter block must keep for each window change only at the start of processing each window, once per line.

The converter must be able to process all the winow layers of each scan line in half the time of an interlaced display line, due to the need to have lines from both fields available in the SRAM for use by the vertical filter block. Therefore the converter needs to operate at a rate at least equal to four times the pixel rate of the interlaced display. For example, with a 13.5 MHz display pixel clock, if the converter operates at 54Mpixels/second it can convert two windows, each covering the width of the display, in half of the active display time of an interlaced scan line. Actually there some more time available, since the active display time leaves out the blanking time, while the converter can operate continuously. Therefore if the converter produces 54Mpixels/second it can support the requirements of two full width windows plus a small pointer window. A converter that operates faster would provide the ability to process more windows per line. If there are times when the vertical filter operation is not needed, the converter and blend blocks would have the full time of a display scan line to complete the composition of a line for display, doubling the amount of work that could be completed.

2.1.6 CLUT

The CLUT is effectively part of the converter block. It is used to convert rows of graphics pixels which are in a CLUT color space. There is one CLUT only in the preferred implementation, although it would be straightforward to

have more than one and to switch between them. The CLUT can be re-written by retrieving new CLUT data via the DMA block when required. Note that it will typically take much longer to re-write the CLUT than the time available in a horizontal blanking interval, so generally the system must allow a whole line time for the Sapphire engine to change the CLUT, although non-CLUT images can be displayed while the CLUT is being changed. The color space of the entries in the CLUT could be either YUV or RGB, however it is preferable from a hardware point of view to store entries in YUV space, while software is a bit simpler if they are stored in RGB space. Note that a color space converter is already required for RGB operation. This choice is currently an open issue.

2.1.7 Blend

The input to the blend block is the YUVa24 output from the converter, with the entire width of one scan line of one window being processed at a time, with the back-most graphics window being processed first. The blend block uses this input to modify the contents of the multi-line buffer that is implemented in the SRAM block. The result of each pixel blend operation is a pixel in the SRAM which consists of the properly weighted sum of the various graphics layers up to and including the present one, and the appropriate alpha blend value for the video layers, taking into account the graphics layers up to an including the present one.

2.1.8 Vertical AFF and SRC

The anti-flutter filter (AFF) and vertical sample rate converter (SRC) are two functions of the same block. This block takes input from the multiple line buffers which are in the SRAM and performs a finite impulse response polyphase filter on the data. The filter acts only in the vertical axis, that is it takes input from vertically adjacent pixels at one time. It multiplies each input pixel times a specified coefficient, and sums the result to produce the output. The polyphase action means that the coefficients, which are samples of a (more or less) continuous impulse response, may be selected from a different fractional-pixel phase of the impulse repsonse every pixel.

2.2 Video display architecture

The video display pipeline portion of the architecture is similar to that of the graphics display pipeline, and it shares some elements with it. The fundamental differences are:

1. The video pipeline supports up to one scaled video window per scan line and one passthrough video window, plus one background color, all of which are logically behind the set of graphics windows. The order of these windows, from back to front, is fixed as background, then passthrough, then scaled video.

2. The video windows are always in YUV format, although they can be in either 4:2:2 or 4:2:0 variants of YUV.

3. The scaled video window can be scaled up in both directions by the display engine, with a factor that can range up to 4. There is not the specific ability nor intent to correct for square pixel aspect ratio as there is for graphics.

4. The scaled video window can be alpha blended into the passthrough and background, using a constant alpha. There is no other blending or chroma keying capability.

The shared elements between the graphics and video display pipelines are the bulk DRAM, the window controller, and the DMA. The FIFOs are logically separate however they can be implemented in a common SRAM, depending on which implementation is smallest in the target technology.

One of the main added features of the video pipeline is the 2-D up-scaler. This is scalar function is a set of two polyphase SRC functions, one for each dimension. The vertical SRC uses a four-tap filter with programmable coefficients (in a fashion similar to the vertical filter in the graphics pipeline), and the horizontal filter uses an 8-tap SRC, again with programmable coefficients. Note that it is possible that a shorter horizontal filter would suffice, such as a 4-tap horizontal SRC for the video up-scaler. Note that since the same filter will be used for downscaling, it may be necessary to use more taps than are strictly needed for up-scaling, since down scaling has more stringent requirements for low pass filtering.

3 Details of architectural blocks

3.1 Window control block

The window control block manages the operation of all of the other blocks in both the graphics and video display pipeline. It obtains, via DMA, descriptors of the various graphics windows which are to be displayed, and information contained in these descriptors is used to control reading of raw graphics data, conversion of this data to the common YUVa format, blending of the various layers, and scaling. The window controller has its own DMA channel to request reading of window descriptors.

Each window descriptor contains the following fields of information:

- Window layer number - 4 bits
- Starting line and pixel of display - 10 + 10 bits
- Ending line and pixel of display - 10 + 10 bits
- Starting address of data

  (Alignment of start address and pitch of pixel maps may be determined later. Alignment to 2 bytes facilitates use of 16 bit wide memory, and 4 bytes for 32 bit memory. Such alignment prevents soft scrolling of graphics by simply changing the start address, especially for 2, 4 and 8 bpp graphics) - 20 to 22 bits
- Pitch of memory addressing - 8 to 12 bits
- Format of data (color space type, etc.) - 4 bits
- Alpha method - 4 bits
- Alpha of the window - 8 bits
- Blanking of 0 to 16 (first and last pixels per line)

  (By indicating that some pixels at the start and end of each line of a window may be blanked, it is possible to

implement soft scrolling of the contents of a window that appears to be stationary on the screen, without requiring the display pipeline to process unaligned pixel addresses. -- 8 bits

- Optional operations e.g. re-load CLUT - 2 or more bits

The total number of bits needed for window descriptors ranges from 98 to 104 or more bits. They should be arranged so that they fit well within a total size of 16 bytes which is 128 bits.

Note that the last entry in the list above implies operations other than the display of windows. This is to allow the window control and DMA mechanism to be used for other important purposes. One example of such a purpose is to read a new set of values into the CLUT at the appropriate time, i.e. after completing the conversion of one CLUT-based window and before the beginning of conversion of the next CLUT-based window. There may be other operations for which this mechanism would be useful, hence more than one bit are included to control such operations.

The window control block has internal memory to store a small number, perhaps 6 or 8, of window descriptors. It needs to store a number equal to the number of windows currently active, plus one, in order to determine that this last window is in fact not currently active. The window descriptors are read from bulk memory automatically by the window control block, via the DMA. The window descriptors must be placed in order in memory before the hardware can perform properly, where "order" means raster scan order of the top left corner of each window. The window control block could easily be constructed to hold a larger number of window descriptors, which could support a large number of sprites per display line, assuming there is enough time to complete processing of every line. Supporting a larger number of sprites via more window descriptor memory in the window controller may be advantageous.

At the start of each display field the window control block clears its internal descriptor memory and begins reading window descriptors into its internal memory, starting at the top of the list of descriptors. It has a pointer

register which points to the top of the descriptor list. Descriptors are read into the window control memory until the most recently read descriptor refers to a window that is not visible on the current line, or the internal memory is full, whichever comes first. The reading and processing of the list of window descriptors repeats for every display field. The line numbers used in the descriptors refer to the complete frame, rather than the lines within a field.

Once the required descriptors are in internal memory, the window control block determines the reverse depth order of the windows and sets up DMA, conversion, and blending operations for the windows in this order, that is, from the back-most window first to the front-most window last. Note that scaling operations cannot, in general, change from one window to the next, and so the scaling parameters are not included in the window descriptors.

It is possible that different scaling parameters could be supported for different windows on the same screen, requiring that the scaling parameters be updated via the window descriptors.

For each window to be processed, this block sets up the DMA block to read bursts of data corresponding to the current line of the window. Since the graphics and video pipelines are separate and can operate in parallel, the window controller may need to set up one DMA operation for each, to operate concurrently. Once the DMA read of one line of graphics or video is complete, it adds the pitch value to the current start-of-line address to prepare for the start of the next line for this window. The window control indicates the data format type to the conversion block, thus enabling the converter to interpret properly the graphics data. This information includes the means of specifying the alpha values, which may be a constant per window, alpha per pixel, alpha per CLUT entry, or derivation from the Y value of each pixel. The window control block indicates to the blending block the start and end pixel locations of the current window, and the blender uses this information to process the converted graphics pixels and write the result into the line buffer SRAM.

Once one complete line of one window of graphics is read and processed (or in the pipeline to be processed) the

window controller repeats the process for the next window, until all graphics windows for the current line have been processed.

Care must be taken to ensure that all blocks in each pipeline have correct information concerning the start and end pixel locations of each window, to ensure that the operations are correctly synchronized. One suggested approach[10] is to have the window controller indicate the pertinent information to all blocks in the pipeline and then allow each block to operate as a self-contained element in a stallable pipeline. That is, if a block needs data from a block which feeds it, it requests data, and it may have to wait for data. If a block has data to send to a subsequent block it may have to wait if the recipient is not ready for (requesting) data. One important effort in specifying the interfaces between the various blocks is the determination of the best way to indicate such states as "ready to receive data" and "ready to send data" without introducing unnecessary pipeline delays or wasted cycles.

3.2 DMA

The DMA block is shared by both paths of the display pipeline as well as some of the control logic, such as the window controller and the CLUT. In practice the DMA block may be shared with other functions on the chip which includes this display engine, functions which cannot be specified here. This document is written as if the DMA block is supporting only the display engine, while we recognize that in practice it will have more channels and more activity, implying among other things more complex real-time behavior.

There are some key choices in the specification of the DMA block; such that various answers would be acceptable. For example, does the DMA block treat each rectangular window (an array of non-contiguous lines of memory) as one unit, needing only a series of requests to read all of it, or does it treat each line as a basic unit, requiring another block such as the window control block to initiate the transfer of each line separately? My current preference would be to keep the DMA block simple, and have it deal in units of lines, not windows, and require another block to manage the setup of each line. This allows the DMA block to

scale up to more meet more complex requirements or even to be replaced with another company's DAM block without requiring excessive re-engineering. Also, is the DMA block responsible for detecting the crossing of page boundaries in DRAM? Or is this the responsibility of the memory controller, or of another block which sets up the DMA engine? My preference here would be to have the DMA block have this responsibility, in order to simplify the memory controller and still not require other blocks which are not directly associated with the memory to be dependent on the specific DRAM architecture.

In order to support the display pipeline, the DMA block needs four channels:

1. Window descriptor read
2. Graphics pixel read
3. Scaled video Y read
4. Scaled video UV read

The graphics read channel is used for auxiliary functions such as reading a new CLUT from DRAM.

Each channel has externally accessible registers to control the start address and the number of words to read. Note that here we assume that all memory reads (and writes) are on word (32 bit) boundaries. This requires that any special cases, such as starting a display window with an odd numbered one-byte pixel, be handled by other means, such as starting a virtual display window on the nearest full word boundary and blanking the unwanted pixels from the beginning of the window in order to achieve the desired alignment. The same can be done for the end of each window. Forcing all addresses handled by the DMA block to be word-aligned simplifies the design of much of the data path, and it reduces the number of bits required for address and pitch.

Individual memory bursts are requested by the block which will receive (or produce) the data. In the case of the display pipeline, these channel requesters are either the FIFOs, for graphics and scaled video pixels, or the window controller, for window descriptors or CLUT data.

The DMA block includes the arbiter, which arbitrates between requests from various requesters, each associated

with a specific DMA channel. The arbitration algorithm will be chosen to guarantee correct real-time performance; it will likely be either a strict priority system or a combination of priority and a fairness algorithm such as a round-robin between channels with the same priority.

Once the DMA block has completed a transfer, as indicated by its start and length registers, it activates a signal that indicates that the transfer is complete, which lets the block which sets up operations for that channel to begin the setup of another transfer. In the case of graphics and video pixel reads, this means that the window controller sets up a transfer of one line of pixels and then waits for the DMA controller to indicate that the transfer of that line is complete before setting up the transfer of the next line, or of a line of another window.

3.3 FIFO

The FIFOs in the display path accept data as they are read from memory, at the full memory data rate and using the clock of the memory controller, and feed this data to subsequent blocks in the video and graphics pipelines. There are three logical FIFOs, one for graphics, one for video Y components, and one for video U and V components. There may be one, two or three physical FIFO memories, depending on which implementation is most efficient in the targeted technology for each implementation. It appears that the most cost-effective implementation may be to share one physical dual-ported SRAM to implement all three of these FIFOs; this requires one write port and probably two read ports, one for graphics and one shared between Y and UV for video; it may be possible to use only one read port. The read ports on the FIFO may operate in a different clock domain from the write port, if the video output logic has a different clock domain from that of the memory controller. The FIFO control logic indicates to the DMA controller when it is ready to receive another burst of data, and it indicates to the subsequent blocks (graphics converter or video SRAM multi-line buffer) when it has data available.

3.4 Graphics converter

Graphics pixels are read from the FIFO in raw graphics format, using one of the multiple formats allowed by the architecture and specified in the window descriptor. Each pixel may occupy as little as 2 bits or as much as 16 bits

(or possibly more if more formats are added). Each pixel is converted to a common YUVa24 format, such as two adjacent pixels share a UV pair and every Y, U, V and a (alpha) component occupies 8 bits. The conversion process is dependent on the pixel format type and the alpha specification method, both of which are indicated by the window descriptor for the currently active window. The graphics converter uses the CLUT memory to convert CLUT format pixels into RGB or YUV pixels.

Conversions of RGB pixels require conversion to YUV, and therefore there is a color space converter within the graphics converter. The color space converter needs a good amount of accuracy for converting RGB16 pixels, such as 7 bit accuracy for all coefficients. If the converter is accurate to 8 or 9 bits it can be used to convert accurately 8 bit per component graphics, such as CLUT entries with this level or accuracy (which may is probably not required in most implementations) or RGB24 images, which are not required in the early implementations but which may become a requirement in future versions.

The graphics converter produces one converted pixel per clock cycle, even when there are multiple graphics pixels packed into one word of data from the FIFO. For now we assume that this clock rate is 54MHz in the first implementation, although it could be faster.

The converter reads data from the FIFO whenever both of these conditions are met: the converter is ready to receive more data, and the FIFO has data ready. The converter has an input from the blend block, which is the next block in the pipeline, which indicates when the blender is ready to receive more converted pixel data. The converter may be forced to stall if the blend block is not ready, and as a result it may not be ready to receive data from the FIFO.

3.5 CLUT

The color look up table (CLUT) is closely associated with the graphics converter, or more properly, it is contained within the converter block. It consists of an SRAM with 256 entries and between 24 and 32 bits per entry. Each entry contains three color components - either RGB or YUV format - and an alpha component. For every CLUT-format pixel converted, the pixel data is used as the address to the CLUT and the resulting value is used by the converter to

produce the YUVa pixel value. The CLUT must support a throughput of one pixel per clock cycle of the converter block.

In normal operation the CLUT is written to only under the control of the window controller, that is, when the window controller is processing a descriptor which indicates that a new set of values is to be written to the CLUT. The CLUT can also be written to and read from the CPU bus for testing.

3.6 Graphics blender

Each graphics pixel which is produced by the converter is blended into the graphics line buffer SRAM by this block.

During the composition of the first window to the SRAM, all pixels which are not part of this window are initialized to a value that represents transparent pixels, i.e. to super-black color and the complement of transparent (Y=0, U=V=128, and alpha =0xFF). For the first (back) graphics window, the graphics pixels are multiplied by their respective alpha values, and the result is written into the buffer. The alpha value is complemented to produce (1-alpha), and that result is written into the alpha portion of the buffer for each pixel. Mathematically this is:

```
buffer_YUV = graphics_pixel // if there is one, or
           = super_black // if there isn't

buffer_alpha = (1 - graphics_alpha) // if there is a graphics pixel, or
             = (1 - 0) = 1 // if there are no graphics in this window at this pixel
```

For subsequent windows, the graphics data which are already in the buffer are used as a source for blending with the new graphics window. The buffer contents then become:

```
new_buffer_YUV = (1 - new_alpha) * old_buffer_YUV + new_alpha * new_pixel
```

for the YUV data, and the alpha component becomes:

```
new_buffer_alpha = (1 - new_alpha) * old_buffer_alpha
```

This process repeats until all the graphics layers have been processed. Since in this spec there is a maximum of three graphics layers on any one line, the entire process runs up to three times, including the back window, although in principle it could run as many times as can be completed in the time available, which is one-half of the display time of a TV scan line. When it is complete, the YUV portion of the line buffer contains the alpha-multiplied composition of all the graphics layers, and the alpha portion contains the coefficient which will be used to multiply times the video layer or layers to produce the correct alpha blending function when the product of the video times its alpha is added to the graphics contents of the line buffer.

Since the line buffer SRAM stores converted and blended pixels in YUVa24 format, with 4:2:2:4 sampling, the U and V components are subsampled. The subsampling of U and V must be handled correctly in the blending block. This means that in most cases a pair of YUVa pixels from the converter are blended into a corresponding pair of YUVa pixels in the SRAM. In so doing, two UV pairs, one from each of the two pairs of pixels, are blended together and the result is written back to the SRAM. At the left and right edges of windows, there may be a pixel which has an odd numbered location on the screen. In order to blend such pixels correctly, the UV pair from the source (from the converter) must be blended with the UV value from the SRAM using the appropriate weighting, that is, only half as much weight is given to the new pixel's UV pair as would be given to the UV from an aligned pair of two YUV pixels. Other approaches would include pre-multiplying the UV by a factor such as one-half when creating the YUVa values in the converter.

Due to the need to process UV components for every pair of pixels, it appears that the best way to implement the blending function is to read pairs of pixels from the SRAM, then process both pixels, then write the resulting pair back to the SRAM.

An additional function that could be very valuable in the graphics blending block would be to blend the top an bottom edges of graphics windows with an alpha value of about half that which would otherwise be used within the graphics

window. This blending provides an anti-flutter effect at the edges of graphics windows, without requiring the use of the vertical AFF. In some cases graphics images will be constructed in such a way that they have already undergone an optimized AFF function, and so they would not need any additional AFF except at the top and bottom edges of the window. These edges may however be the cause of very disturbing flutter which may be unavoidable by another technique. Addressing this problem in this way can produce superior visual results to those obtained with the use of a standard AFF later in the pipeline. The implementation of this special case blending of the top and bottom edges of graphics windows is simply to divide the alpha that would otherwise be used for these pixels by 2 before blending the pixels normally. This division of alpha by 2 could be implemented in either the converter or the blender block. In either case, the responsible block needs to be aware of which lines form the top and bottom lines of each graphics window. Special blending of the top and bottom edges of graphics windows may be added, and consideration of which block would be best suited to modify alpha.

The performance required of the blend block is similar to the performance required of the converter block. However, each pixel processed by the blend block requires both a read from and a write to the line buffer SRAM. Using the same arguments raised above, the minimum throughput of the blend block is 54Mpixels/second, to support two full windows and one small pointer window. This requires pipelining the read, blend, and write operations if this througput is to be obtained using a clock speed of 54MHz. Alternatively, a clock speed of 108MHz could be used to perform separate reads and writes.

The processing of each pixel requires a significant number of multiplies and adds:

- Multiply each Y, U, and V component of the current window by alpha: 2 multiplies per pixel
- Multiply each Y, U and V component from the SRAM by its alpha: 2 multiplies per pixel
- Add Y, U and V from both of the above products: 2 adds per pixel

- Multiply the alpha from the SRAM by (1-alpha) from the current window: one subtract and one multiply.

The graphics blend block addresses the line buffer SRAM directly, rather than treating it as a set of line buffers. This direct connection enables the blender to read and write pixels directly as it constructs each line using each of the windows, one layer at a time. The blender receives inputs from the vertical graphics filter indicating when a line buffer within the SRAM is no longer needed by the filter and is therefore available for use by the blender to begin constructing the next display line. The blender indicates to the graphics converter when it is ready to receive more pixel data. The blender can stall when the line buffer memory is not available, and as a result the preceding elements in the pipeline may stall as well.

3.7 Graphics line buffer SRAM

The graphics line buffer SRAM needs to store the same number of lines of YUVa24 graphics pixels as are used by the subsequent vertical filter. Assuming that the vertical filter uses four taps, the SRAM then requires a capacity of at least 720 pixels/line * 3 bytes/pixel * 4 lines = 8,640 bytes. Each of the ports to this memory is 24 bits wide, to accommodate the YUVa24 common intermediate format. If we choose to use a 2 tap vertical filter the size of this SRAM could be reduced by half to 4,320 bytes. The actual size may have to be somewhat larger to accommodate the sizes available in the target technology. The SRAM could be split into multiple memories with narrower widths if that would better fit the technology.

The SRAM needs at least 2 read ports and one write port: One read and one write for the blender interface, which needs to perform a read-modify-write on average once per clock cycle, and one read port could be used for all four (two) of the inputs to the vertical filter. With an output pixel sample rate of 13.5MHz, a clock rate of four times this or 54MHz would suffice for a 4 tap vertical filter.

3.8 Graphics vertical AFF and SRC

The vertical AFF and SRC a finite impulse response polyphase filter operating on the vertical axis of graphics pixels. The purpose of the low pass filter is to minimize or eliminate visible vertical motion on the screen which is

the result of aliasing high vertical frequency components, particularly those around one-half the vertical sample rate, into temporal components. The purpose of the SRC is to change the vertical sample rate to adjust graphics images which are design with square aspect ratio pixels so that they appear with the correct aspect ratio on televisions with 720 samples per line on 4:3 displays.

The AFF function can be obtained by a variety of low pass filter characteristics. The simplest, and one which is known to be highly effective, is simply to average two vertically adjacent lines (from the complete frame, not from a single field) to produce the output for one field. This averaging of two lines causes a frequency null at half the vertical sample frequency, which addresses the frequency domain requirement, and it also can be shown to make horizontal edges of large areas to appear in a stationary position across both interlaced fields. Such a filter causes a half pixel vertical phase shift, since it has an even number of taps. While it works very well for horizontal edges of large areas, it is not optimal for diagonal edges (it leaves them a little bit more blurry looking than some other filters) nor is it optimal for single-line features such as horizontal single lines and single dots. Nevertheless it is a likely candidate for a high quality AFF.

The SRC function adjusts pixel maps which are designed for square pixel display so that they look correct on displays with non-square pixel sampling. This function can be obtained either by horizontal scaling or vertical scaling (or in theory by a combination of the two). For NTSC displays, there are about 480 horizontal lines, and for the standard 4:3 display aspect ratio the largest square pixel image is 640 x 480 (note that 640 = 480 * (4/3)). Using the video standard of 13.5 MHz sampling, there are 720 (or 704, depending on which definition you take) pixels on every line, occupying the same amount of space on the screen as the 640 samples of a square pixel display would. Therefore the 640 x 480 image can be made to appear with the correct square pixel aspect ratio and also fill the entire screen by scaling the image up horizontally by a factor of 720/640 or 704/640.

Alternatively, the horizontal size can be left unmodified at 640, such that it occupies 640 / 720 = 89% of the screen width, and the vertical size can be scaled down accordingly

to 89% of its original height, that is, the result is 426 or 428 lines high. Using the vertical method makes the 640 x 480 fit within an area just slightly larger than what is usually considered the safe display area on an NTSC TV, such that most of the image is guaranteed to be visible, taking into account the overscan of the typical TV, and on some TVs the entire image and some of the background around it will be visible.

For PAL and SECAM TVs, with 625 lines total per frame period and 50 frames per second, there are about 576 displayable horizontal lines per frame, while the number of samples per line remains 720 (not 704). Again, there are two choices. A full width image made up of square pixels has a size of 768 x 576 (576 * (4/3) = 768), and such images could be scaled down horizontally to a size of 720 x 576, with the result that the image fills the entire displayable area. The vertical scaling solution allows the use of the same 640 x 480 image size, and just as in the NTSC case it leaves the width unmodified at 640 pixels, such that it covers 89% of the width. In the PAL case, the height of the image is scaled up to 89% of the full 576 line frame height, or 512 lines.

We have chosen to implement the vertical scaling technique to address the problem of square pixel graphics. This allows the use of a common 640 x 480 image size regardless of the TV standard used, and it does a reasonable job of fitting the image onto the displayable area of normal televisions, rather than losing much of it to overscan. It also does not increase the memory requirements for PAL applications, which can be a serious issue. By not scaling horizontally we do not lose image quality in the horizontal axis, which is where more of the detail of computer imagery tends to exist and where the resolution of the best TVs is highest. Also, since we need a vertical filter anyway in order to implement the AFF, we do not need to introduce an additional filter block to perform aspect ratio correction.

While it appears that the vertical filter requires four taps, this has not yet been firmly established; it is possible that two taps could suffice. Note that in general an even number of taps is required to implement a polyphase SRC. If it were not for the SRC function, two taps would probably provide an adequate or even preferable quality of AFF, while four taps apparently give a better SRC function. Two taps requires a minimum of two line buffer in the SRAM,

while four taps requires a minimum of four line buffers in the SRAM. The reason that the number of lines that must be stored in the SRAM is equal to the number of taps, rather than one less than the number of taps, is that the blend block builds up a scan line over multiple passes, and the final line is only available from memory, after the blend process for a line is complete.

The multiple input taps are produced by the filter reading one pixel from each of the 4 (or 2) lines sequentially. The SRAM and the logic to read it therefore runs at a minimum of 4 (2) times the output pixel rate, or 54MHz (27MHz).

The vertical filter coefficients are read from a coefficient memory. This allows software to install any desired filter characteristic, at the expense of the silicon needed to store the coefficients and the full multipliers required to implement arbitrary coefficients. We believe that the correct number of phases for the polyphase function is four, meaning that there are four coefficients for every tap. There are either two or four taps in the filter. Therefore the coefficient memory would appear to need to store 2 * 4 or 4 * 4 coefficients. However, it may be possible to economize on this storage, since the filter kernels are always symmetrical. Typical filter kernels for low pass and SRC function are illustrated in Fig. 3:

Fig. 3 shows what is refered to herein as a four tap filter kernel; it is drawn to represent a Gaussian shape. Note that there are five sample positions drawn, although the outer two have values of zero. When used for a sample rate converter, for any phase but zero phase output the coefficients used are taken from intermediate points between the sample positions marked, and there will always be four non-zero coefficients. When the phase setting is zero, there are three non-zero coefficients. The main thing to notice, for reasons of hardware architecture, about this being called a "Gaussian" filter is that it is not made up of straight lines, that is, the coefficients cannot be determined simply by interpolating along straight lines, as would be the case for simpler filters. If the coefficients are retrieved from memory they could desribe almost any shape. If the coefficients were restricted to follow straight lines from zero to the center value and back, the coefficients could be calculated directly without being stored at all. In cases where the phase adjustment is set

to zero and there are three, symmetrically valued non-zero coefficients, the phase shift of the filter is zero while there is some low-pass filter effect. This property helps such a filter design produce a consistent AFF (low pas filter) function while simultaneously operating as a polyphase filter for SRC purposes.

Using a bi-linear two-tap filter kernel, as shown in Fig. 4, there are two non-zero coefficients for phase settings other than zero, and only one non-zero coefficient when the phase is zero, and its value is 1.0. When using such a filter as an AFF, the phase is always set to one-half of a sample, such that the two coefficients are both equal to one-half. Such a filter can be represented as a two-tap rectangle, rather than the triangular shape drawn. The rectangle does not have a tap in the middle, rather it simply has two taps. One side effect of such a filter is that is always has a phase shift of one-half a sample, since it is asymmetrical.

The two tap rectangular filter has the attractive property that it has a zero in its frequency response at half the sample rate (this is readily observed by examining what happens when a function of alternating ones and zeros is used as the input), and this null eliminates the most common cause of flutter in interlaced displays. Such flutter is commonly caused by vertical axis frequencies at or above half the vertical sample rate which alias to the frame rate at zero spatial frequency.

A four tap filter that is to be used as an AFF should also be designed to have a null at half the vertical sample rate. This is readily achieved. One way to do this is by creating the filter kernel by convolving a 2 tap rectangular function with some other function to produce the full 4 tap filter kernel. The 2 tap rectangular function causes the desired zero.

One important effect of the vertical filter block is to filter the alpha values correctly. The alpha values that are found in the line buffer SRAM and used as input to the filter block are the alpha values that would be multiplied by the video layer (which itself is a composite of other images) combine with the graphics, if there were no vertical filter. The graphics pixels found in the line buffers have been pre-mulitplied by alpha, so no further multiplication of them by any alpha value is necessary.

When performing a low pass filter or sample rate conversion on the graphics pixels, the alpha values should be filtered as well. The most straightforward solution is to perform the same filter function on the alpha as is performed on the graphics. It may be possible, however, to implement a 2-tap filter and SRC on the alpha values even when the graphics filter has 4 taps, if such an implementation would result in considerable savings in cost, complexity and risk.

3.8.1 Potential horizontal graphics LPF

As a separate matter, there may be some value in adding a horizontal low pass filter to the graphics display pipeline. This would provide the needed amount of horizontal filtering of graphics which potentially have a great deal of high frequency content, without filtering the video signals which presumably already have the correct spectral characteristics and should not be filtered further. Low pass filtering of graphics is usually performed with a DENC, and if we are to include an optimized DENC we could move the function to the graphics display pipeline and eliminate it from the DENC.

3.9 Video line buffer SRAM

Needs to store 3 or 4 lines of YUV16 video. Alpha does not need to be stored in this memory because there is a constant alpha for the scaled video window. It is possible to store only 3 lines in memory since the fourth line could be read in real time from the DRAM. Therefore the minimum size of this SRAM is 4,320 bytes, with a 16 bit width.

Since the video scaler block and the SRAM can be used either for up-scaling the output or down scaling the input of the scaled video window, the input to the SRAM can come from a video capture port instead of the DRAM. In this mode, the output of the video scaler is written to the memory rather than blended with other display data streams and fed to the output. Nevertheless, the scaled video output data path must be available even when the scaler and memory are being used for downscaling the input, to display the scaled video image. While it might appear that this is not necessary since the graphics display pipeline can display YUV images, the graphics display is in some cases being scaled and filtered by the graphics vertical AFF and SRC, which is undesirable for the video widows. Also, the

scaled video window can be in YUV 4:2:0 mode, which is not supported by the graphics display pipeline.

3.10 Video scaler

The video scaler consists of a 4 tap polyphase vertical SRC and an 8 tap polyphase horizontal SRC. All of the coefficients are presumed to be programmable, although we can consider making them more limited or fixed. The vertical SRC should support at least 4 phases, while the horizontal SRC should support 8 phases.

3.11 Video - graphics blender

This block blends the passthrough video and the background color with the scaled video window, using the alpha value which is associated with the scaled video window. The result of this blending operation is then blended with the output of the graphics display pipeline. The graphics output has been pre-multiplied by each of its pixels' respective alpha values, and the graphics output contains the correct alpha value for multiplication by the video output. The output of the video blend function is multiplied by this video alpha which is obtained from the graphics pipeline and the resulting video and graphics pixel data streams are added together to produce the final blended result.

4 Blending of Independent Image Layers

4.1 Description

Disclosed in a method of blending multiple layers of imagery in a digital system such as a graphics display devices. The method achieves a visual effect that is similar to, or the same as, successively blending layers from the back layer to the front layer using "alpha" values for every layer.

The desired effect is as follows: For every layer {L1, L2, L3…Ln}, with L1 being the back-most layer, each layer is blended with the one behind it, beginning with L2 being

blended on top of L1. The intermediate result R(i) from the blending of pixels P(i) of layer L(i) over the pixels P(i-1) of layer L(i-1) using alpha value A(i) is:
R(i) = A(i) * P(i) + (1 - Ai) * P(i-1)

The alpha values {A(i)} are in general different for every layer and for every pixel of every layer.
However, in some important applications it is not practical to apply this formula directly, since some layers may need to be processed in spatial dimensions (e.g. 2 dimensional filtering or scaling) before they can be blended with the layer or layers behind them. While it is of course possible to blend the layers first and then perform the spatial processing, that would result in processing the layers behind the subject layer, which may be undesirable. Processing the subject layer first would generally require local storage of all of the pixels required for such processing, which may be prohibitively expensive. This problem is significantly exacerbated when there are multiple layers to be processed above one or more layers that are not to be processed. In order to implement the formula above directly, each of the layers would have to be processed first, i.e. using their own local storage and individual processing, before they could be blended with the layer behind.

The disclosed solution solves this problem in a different way. Rather than blending all the layers from back to front, all of the layers that are to be processed (e.g. filtered) are layered together first, even if there is one or more layers behind them over which they should be blended, and the combined upper layers are then blended with the back-more layers that are not to be processed. For example, layers {1, 2 and 3} may be layers that are not to be processed, while layers {4, 5, 6, 7, and 8} may be layers that are to undergo processing, while all 8 layers are to be blended together, using {A(i)} values that are independent for every layer and pixel.

The solution is as follows: All of the layers that are to be filtering (referred to as the "upper" layers) are blended together from back to front (remember that the back-most of the upper layers is not in general the back-most layer of the entire operation), using a partial blending operation. At each stage of this blending, a intermediate alpha value is maintained for later use for

blending with the layers that are not to be filtered (referred to as the "lower" layers).

The preferred form of the formula is:

$$R(i) = A(i) * P(i) + (1 - Ai) * P(i-1)$$

and

$$AR(i) = AR(i-1) * (1 - A(i))$$

where P(i-1) initially represents black before any layers are blended, AR(i) is the alpha value resulting from each instance of this operation, and AR(i-1) initially represents transparency before any layers are blended. AR represents the alpha value that will subsequently be multiplied by the lower layers as indicated below, and so an AR value of 1 (assuming alpha ranges from 0 to 1) indicates that the current pixel is transparent, not opaque, and the lower layers will be fully visible when multiplied by 1.

In other words, at each stage of blending the upper layers, the pixels of the current layer are blended into a using the current alpha value, and also an intermediate alpha value is calculated as the product of (1-A(i)) * (1-A(i-1)). The key differences between this and the direct evaluation of the conventional formula are: (1) the calculation of the products of (1-A(i)) for each of the upper layers, and (2) a virtual transparent black layer is used to initialize the process for blending the upper layers, since the lower layers that would normally be blended with the upper layers are not used yet in this process.

To complete the blending process of the entire series of layers, including the upper and lower layers, once the upper layers have been blended together as described above, they can be processed as desired (the resulting alpha values can also be processed as desired), and then the result of this processing, a composite intermediate image, is blended with the lower layer or layers. The lower layers can be blended in the conventional fashion, so at some point there can be a single image representing the lower layers. Therefore two images, one representing the upper layers and one representing the lower layers can be blended together. In this operating, the AR(n) value at each pixel that results from the blending of the upper layers and any subsequent processing is used to multiply times the composite lower layer.

Mathematically this latter operation is as follows: Let L(u) be the composite upper layer resulting from the process described above and after any processing, let AR(u) be the composite alpha value of the upper layers resulting from the process above and after any processing, let L(l) be the composite lower layer that results from blending all lower layers in the conventional fashion and after any processing, and let Result be the final result of blending all the upper and lower layers, after any processing. Then,
Result = L(u) + AR(u) * L(l)
Note that L(u) does not need to be multiplied by any additional alpha values, since all such multiplications were already performed at an earlier stage.

In a preferred embodiment, a series of images makes up the upper layers. These are created by reading pixels from memory, as in a conventional graphics display device. Each pixel is converted into a common format if it is not already in that format; in this example the YUV format is used (also called Y, Cb, Cr, as defined by the standard ITU-R 601B). Each pixel also has an alpha value associated with it. The alpha values can come from a variety of sources, including (1) being part of the pixel value read from memory (2) an element in a color look-up table (CLUT) in cases where the pixel format uses a CLUT (3) calculated from the pixel color value, e.g. alpha as a function of Y, (4) calculated using a keying function, i.e. some pixel values are transparent (i.e. alpha = 0) and others are opaque (alpha = 1) based on a comparison of the pixel value with a set of reference values, (5) an alpha value may be associated with a region of the image as described externally, such as a rectangular region, described by the four corners of the rectangle, may have a single alpha value associated with it, or (6) some combination of these.

The upper layers are composited in the memory storage buffers called "line buffers". There is one line buffer associated with each scan line to be composited. Each line buffer has an element for each pixel on a line, and each pixel in the line buffer has elements for the color components, in this case Y, U and V, and one for the intermediate alpha value AR. Before compositing of each line begins, the appropriate line buffer is initialized to represent a transparent black having already been composited into the buffer; that is, the YUV value is set to the value that represents black (i.e. Y = 0, U = V =

128) and the alpha value AR is set to represent (1-transparent) = (1-0) = 1.
Each pixel of the current layer on the current line is combined with the value pre-existing in the line buffer using the formulas already described, i.e.

$$R(i) = A(i) * P(i) + (1 - A(i)) * P(i-1)$$

and

$$AR(i) = AR(i-1) * (1 - A(i))$$

That is, the color value of the current pixel P(i) is multiplied by its alpha value A(i), and the pixel in the line buffer representing the same location on the line P(i-1) is read from the line buffer, multiplied by (1-A(i)), and added to the previous result, producing the resulting pixel value R(i). Also, the alpha value at the same location in the line buffer (AR(i-1)) is read from the buffer and multiplied by (1-A(i)), producing AR(i). The results R(i) and AR(i) are then written back to the line buffer in the same location.

Note that when multiplying a YUV value by a alpha value between 0 and 1 the offset nature of the U and V values must be accounted for. That is, U = V = 128 represents a lack of color and it is the value that should result from a YUV color value being multiplied by 0. This can be done in at least two ways. The preferred method is to subtract 128 from the U and V values before multiplying by alpha, and then add 128 to the result. Another method is to multiply the U and V values directly by alpha, and ensure that at the end of the entire compositing process all of the coefficients multiplied by U and V sum to 1, so that the offset 128 value is not distorted significantly.

Each of the layers in the group of upper layers is composited into the line buffer starting with the back-most of the upper layers and progressing towards the front until the front-most of the upper layers has been composited into the line buffer. In this way, a single hardware block (the "compositor") is used to implement the formula above for all of the upper layers. This requires the compositor to operate at a clock frequency that is substantially higher than the pixel display rate. In the preferred embodiment the compositor operates at 81MHz while the pixel display rate is 13.5MHz.

This process repeats for all of the lines in the entire image, starting at the top scan line and progressing to the bottom.
Once the compositing into each scan line has been completed, the scan line becomes available for use in processing such as filtering or scaling. Such processing can be performed while subsequent scan lines are being composited into other line buffers. Various processing operations may be selected. One operation is a low pass filter in the vertical dimension; this may be performed in order to minimize the "flutter" effect inherent in interlaced displays such as television.

Another operation is a vertical downscaling or upscaling; this may be performed in order to change the pixel aspect ratio from the square pixels that are normal for computer, Internet and World Wide Web content into any of the various oblong aspect ratios that are standard for televisions as specified in ITU-R 601B. In order to be able to perform vertical scaling of the upper layers in this embodiment, there are seven line buffers total. This allows for 4 line buffers to be used for filtering and scaling, 2 are available for progressing by 1 or 2 lines at the end of every line, and 1 for the current compositing operation. When scaling or filtering are performed, the alpha values in the line buffers are filtered or scaling in the same way as the YUV values, ensuring the resulting alpha values correctly represent the desired alpha values at the proper location. Either or both of these operations, or neither, or other processing, may be performed on the contents of the line buffers.

Once the optional processing of the contents of the line buffers has been completed, the result is the completed set of upper layers with the associated alpha value (product of (1-A(i)). These results are used directly for compositing the upper layers with the lower layers, using the formula Result = L(u) + AR(u) * L(l). If the lower layers require any processing independent of processing required for the upper layers or for the resulting image, the lower layers are processed before being combined with the upper layers; however in the preferred embodiment no such processing is required.

Each of the operations described above in the preferred embodiment is implemented digitally using conventional ASIC technology. As part of the normal ASIC technology the

logical operations are segmented into pipeline stages, which require temporary storage of logic values from one clock cycle to the next. The choice of how many pipeline stages are used in each of the operations described above is dependent on the specific ASIC technology used, the clock speed chosen, the design tools used, and the preference of the designer, and may vary without loss of generality. In the preferred embodiment the line buffers are implemented as dual port memories allowing one read and one write cycle to occur simultaneously, facilitating the read and write operations described above while maintaining a clock frequency of 81MHz. In this embodiment the compositing function is divided into multiple pipeline stages, and therefore the address being read from the memory is different from the address being written to the same memory during the same clock cycle.

Each of the arithmetic operations described above in the preferred embodiment use 8 bit accuracy for each operand; this is sufficient for the required accuracy of the final result. Products are rounded to 8 bits before the result is used in subsequent additions.

In an alternative embodiment, line buffers are not used, but rather all of the upper layers is composited concurrently. In this case, there is one compositor hardware block for each of the upper layers, and the clock rate of the compositors can be approximately equal to the pixel display rate (it can be somewhat slower or faster, if FIFO buffers are used at the output of the compositing function.) The mathematical formulas implemented are the same as in the preferred embodiment. The major difference is that instead of performing the compositing function iteratively by reading and writing a line buffer, all layers are composited concurrently and the result of the series of compositor blocks is immediately available for processing, if required, and for blending with the lower layers, and line buffers are not required for purposes of compositing. Line buffers may still be required in order to implement vertical filtering or vertical scaling, as those operations typically require more than one line of the group of upper layers to be available simultaneously, although fewer line buffers are required here than are required for the preferred embodiment. Using multiple compositors operating at approximately the pixel rate simplifies the implementation in applications where the pixel rate is relatively fast for the ASIC technology

used, for example in HDTV video and graphics systems where the pixel rate is 74.25MHz.

5 Method for Interlace Flutter Reduction via Automatic Blending

5.1 Problem

Interlaced displays, such as televisions, have an inherent tendency to display an apparent vertical motion at the horizontal edges of displayed objects, with horizontal lines, and on other points on the display where there is a sharp contrast gradient along the vertical axis. This apparent vertical motion is variously referred to as flutter, flicker, or judder. Some means is required to minimize or eliminate this flutter in order to improve the perceived image quality and to minimize irritation in viewers. While some image elements can be designed specifically for display on interlaced TVs or filtered before they are displayed, when multiple such image objects are combined onto one screen there are still visible flutter artifacts at the horizontal top and bottom edges of these objects. While it is also possible to include filters in hardware to minimize visible flutter of the display, such filters are costly in that they require higher memory bandwidth from the display memory, since both even and odd fields must be read from memory for every display field, and they tend to require additional logic and memory on-chip.

5.2 Solution

The solution is designed for use in graphics displays device that composites visible objects directly onto the screen; for example, the device may use windows, window descriptors and window descriptor lists, or similar mechanisms. The top and bottom edges (first and last scan lines) of each object (or window) are displayed such that the alpha blend factor of these edges is adjusted to be one-half of what it would be if these same lines were not the top and bottom lines of the window. For example, a window may constitute a rectangular shape, and the window may be opaque, i.e. it's alpha blend factor is 1, on a scale of 0 to 1. All lines this window except the first and last are opaque when the window is rendered. The top and bottom lines are adjusted so that, in this case, the alpha factor becomes 0.5, thereby causing these lines to be mixed 50% with the images that are behind them.

This function occurs automatically in the preferred implementation. Since in the preferred implementation windows are rectangular objects that are rendered directly onto the screen, the locations of the top and bottom lines of every window are already known. In the preferred embodiment there exists also the ability to alpha blend each window with the windows behind it, and this alpha value can be adjusted for every pixel, and therefore for every scan line. These characteristics of the application design are used advantageously, as the flutter reduction effect is implemented by controlling the alpha blend function using information that is readily available from the window control logic.

In a specific illustrative example, the window is solid opaque white, and the image behind it is solid opaque black. In the absence of the disclosed method, at the top and bottom edges of the window there would be a sharp contrast between black and white, and when displayed on an interlaced TV, significant flutter would be visible. Using the disclosed method, the top and bottom lines are blended 50% with the background, resulting in a color that is halfway between black and white, or gray. When displayed on an interlaced TV, the apparent visual location of the top and bottom edges of the object is constant, and flutter is not apparent. The same effect applies equally well for other image examples.

Note that this method does not require any increase in memory bandwidth, as the alternate field (the one not currently being displayed) is not read from memory, and there is no need for vertical filtering, which would have required logic and on-chip memory.

The same function can alternatively be implemented in different graphics hardware designs. For example in designs using a frame buffer (conventional design), graphic objects can be composited into the frame buffer with an alpha blend value that is adjusted to one-half of its normal value at the top and bottom edges of each object. Such blending can be performed in software or in a blitter that has a blending capability.

6 Window Soft Horizontal Scrolling Mechanism

6.1 The problem

Soft scrolling of a graphics window (surface) horizontally is difficult to implement due to the following reasons:

1. Graphics memory buffers are usually implemented using low-cost DRAM, SDRAM, for example. But such devices are usually slow and require each burst transfer to be within a page. Smooth or "soft" horizontal scrolling, however, requires the ability to set the starting address to any arbitrary pixel, and this conflicts with the requirement to transfer data in bursts within the well-defined pages of DRAM.

2. Complex control logic would be required to monitor if page boundaries are to be crossed during the transfer of pixel maps for each step during soft horizontal scrolling. Transfer sizes at the pixel map horizontal boundaries need also to be changed accordingly.

3. It is even more difficult to implement independent scrolling of multiple graphics windows.

6.2 The solution

The method disclosed for positioning the contents of graphics windows on arbitrary positions along the horizontal axis is as follows:

1. Each graphics window is addressed using an integer word address. For example, if the memory system uses 32 bit words, then the address of the start of a window is defined to be aligned to a multiple of 32 bits, even if the first pixel that is desired to be displayed is not so aligned.

2. Each graphics window also has associated with it a horizontal offset value, in units of pixels, that indicates a number of pixels to be ignored, starting at the indicated starting address, before the active display of the window starts. For example, if the memory system uses 32 bit words and the graphics format of a specific window uses 8 bits per pixel, the display of the window might ignore one, two or three pixels (8, 16, or 24 bits), causing an effective left shift of 1, 2, or 3 pixels. In this example, a shift of 4 pixels is implemented by changing the start address by one 32-bit

word. Shifts of any number of pixels are thereby implemented by a combination of adjusting the starting word address and adjusting the pixel shift amount. The same mechanism is used for any number of bits per pixel (1, 2, 4, etc.) and any memory word size.

Our solution for soft horizontal scrolling of graphics windows is based on the fact that each graphics window (or surface) is independently stored in a normal graphics buffer memory device (SDRAM, EDO-DRAM, DRAM) as a separate object. Windows are composed on top of each other in real time as required. If a window is to be scrolled (right or left), a special field is defined in the window descriptor that tells how many pixels are to be shifted to the right or the left. This way, a graphics window can be left intact and each time the window is scrolled, only the parameter needs to be changed together with the parameter that defines the size of the window.

The corresponding transfers to the boundaries of a pixel map do not have to be treated differently as the other portions of the pixel map. The complete pixel map is transferred and the left or right edges are truncated or masked on chip according to the amount of the scrolling. This solves the problem that pixel data transfer has to be within a page boundary because we can always allocate the memory such that a pixel map starts at a page boundary. The number of pixels discarded is usually small because the amount to be masked is less than the page size and once the scrolling crosses a page boundary, the window start address can be changed instead of the amount of the scrolling (blank pixels). From the hardware point of view, this helps to reduce the control requirement and the logic associated with it.

Each window can be scrolled independent of each other because each window has its own window descriptor and windows are composed in real-time when they are displayed. So window refreshing and fine (soft) window scrolling can be achieved.

This soft horizontal window scrolling mechanism:

has no special need for transferring boundary pixels differently;

all windows can be scrolled independently;

hardware control is minimal;

very fine scrolling can be achieved;

scrolling is done through pixel map masking or blanking rather than the traditional way, pixel map addressing; and

there is no need for describing pixel maps of various formats differently.

7 Window Descriptor and Solid Surface Description

7.1 The problem

Often in the creation of graphics displays the artist or application developer has a need to includes rectangular objects on the screen, with the objects having a solid color and a uniform alpha blend factor. These regions (or objects) may be rendered with other displayed objects on top of them or beneath them. In conventional graphics devices, such solid color objects are rendered using the number of distinct pixels required to fill the region. It is advantageous in terms of memory size and memory bandwidth to render such objects on the display directly, without expending the memory size or bandwidth required in conventional approaches.

7.2 The solution

In the disclosed solution, the graphics display hardware implements rectangular regions of many types directly in hardware. These regions are variously referred to as windows, surfaces, sprites, or canvasses, with essentially the same meaning. Here we refer to them as windows. Each of the windows on the screen can be defined to have its own value of various parameters, such as location on the screen, starting address in memory, depth order on the screen, pixel color type, etc. Windows can be displayed such that windows may overlap or cover each other, with arbitrary spatial relationships. The parameters that define and control each window are contained in a data structure called a window descriptor. All of the elements that make up any given graphics display screen are specified by combining all of the window descriptors of the windows that make up the screen into a window descriptor list. At every display field time, the graphics hardware constructs the display image from the current window descriptor list, and it composites all of the windows in the list into a complete screen image according to the parameters in the window descriptors and the contents of the windows themselves.

With the introduction of window descriptors and real-time composition of graphics windows, we are able to describe a window with a solid color and fixed translucency entirely in a window descriptor by appropriate fields in it. These parameters describe the color and the translucency (alpha) just as if it is a normal graphics window. The only difference is that there is no pixel map associated with this window descriptor. It is the hardware that will generate a pixel map accordingly and performs the blending in real time when the window is to be displayed. That is, a window consisting of a rectangular object on the screen, with this object having a constant color and constant alpha value, is created simply by including a window descriptor in the window descriptor list such that the subject descriptor indicates the color and the alpha value of the window, and a null pixel format, i.e. no pixel values are to be read from memory. Other parameters indicate the window size and location on the screen, allowing the creation of solid color windows any size and location.

In this embodiment, no pixel map required, memory bandwidth requirements are reduced and a window of any size can be displayed.

8 Window List Sorting Method

8.1 Problem Solved

In a graphics display engine that operates on the basis of lists of window descriptors, windows can be specified to overlap one another. At the same time, windows can start and end on any line, and there can be many windows visible on any one line. With one window overlapping another, some means is needed to indicate and control the depth ordering on the display. There is an extremely large number of possible combinations of window starting and ending locations along vertical and horizontal axes and depth order locations. What is needed are means both to specify the depth order of all windows, in the window descriptor list, and to implement the depth ordering correctly while accounting for all windows properly and in a cost effective way.

8.2 Solution

Each window descriptor includes a parameter indicating the depth location of the associated window. The range that is allowed for this parameter can be defined to be almost any useful value. In the preferred embodiment there are 16 possible depth values, ranging from 0 to 15, with 0 being the back-most (deepest, or furthest from the viewer), and 15 being the top or front-most depth. The window descriptors are ordered in the window descriptor list in order of the first display scan line where the window appears. For example if window A spans lines 10 to 20, window B spans lines 12 to 18, and window C spans lines 5 to 20, the order of these descriptors in the list would be {C, A, B}.

In the hardware - a VLSI device - that implements the compositing of the windows on the screen, there is an on-chip memory capable of storing a number of window descriptors. In the preferred implementation this memory can store up to 8 window descriptors on-chip, however the size of this memory could easily be larger or smaller without loss of generality. Window descriptors are read from main memory into the on-chip descriptor memory in order from the start of the list, and stopping when the on-chip memory is full or when the most recently read descriptor describes a window that is not yet visible, that is, its starting line is on a line that has a higher number than the line currently being constructed. Once a window has been displayed and is no longer visible, it is cast out of the on-chip memory and the next descriptor in the list is read from main memory. At any given display line, the order of the window descriptors in the on-chip memory bears no particular relation to the depth order of the windows on the screen.

The hardware that controls the compositing of windows builds up the display in layers, starting from the back-most layer. In the preferred embodiment, the back most layer is layer 0. The hardware performs a quick search of the back-most window descriptor that has not yet been composited, regardless of its location in the on-chip descriptor memory. In the preferred embodiment, this search is performed as follows:

All 8 window descriptors are stored on chip in such a way that the depth order numbers of all of them are available simultaneously. While the depth numbers in the window descriptors are 4 bit numbers, representing 0 to 15, the

on-chip memory has storage for 5 bits for the depth number. Initially the 5 bit for each descriptor is set to 0. The depth order values are compared in a hierarchy of pair-wise comparisons, and the lower of the two depth numbers in each comparison wins the comparison. That is, at the first stage of the test descriptor pairs {0, 1}, {2, 3}, {4, 5}, and {6, 7} are compared, where {0 - 7} represent the 8 descriptors stored in the on-chip memory. This results in 4 depth numbers with associated descriptor numbers. At the next stage two pair-wise comparisons compare {(0, 1), (2, 3)} and {(4, 5), (6, 7)}. Again, each of these results in a depth number of the lower depth order number and the associated descriptor number. At the third stage one pair-wise comparison finds the smallest depth number of all, and its associated descriptor number. This number points the descriptor in the on-chip memory with the lowest depth number, and therefore the greatest depth, and this descriptor is used first to render the associated window on the screen. Once this window has been rendered onto the screen for the current scan line, the fifth bit of the depth number in the on-chip memory is set to 1, thereby ensuring that the depth value number is greater than 15, and as a result this depth number will never again be found to be the back-most window until all windows have been rendered on this scan line, preventing rendering this window twice. Once all the windows have been rendered for a given scan line, the fifth bits of all the on-chip depth numbers are again set to 0; descriptors that describe windows that are no longer visible on the screen are cast out of the on-chip memory; new descriptors are read from memory as required (that is, if all windows in the on-chip memory are now visible, the next descriptor is read from memory, and this repeats until the most recently read descriptor is not yet visible on the screen), and the process of finding the back most descriptor and rendering windows onto the screen repeats.

9 Packet-Based Control Data Passing Mechanism

9.1 Introduction

In systems such as graphics display controllers where multiple types of data are may be output from one module, such as a memory controller subsystem, and used in another subsystem, such as a graphics processing subsystem, it becomes progressively more difficult to support a combination of a large number of different data types,

dynamically varying data transfer rates and FIFO buffers between the producing and consuming modules. The conventional way to address such problems is to design a logic block that understands the varying parameters of the data types in the first module and controls all of the relevant variables in the second module. This is difficult due to variable delays between the two modules, due to the use of FIFOs between them and varying data rate, and due to the complexity of supporting a large number of data types. Disclosed is a method of conveying all of the required control information from a first module, e.g. a memory controller, to a second module, e.g. a graphics processing system, such that all of the relevant variables in the second module are properly controlled in a timely fashion and such that the control is not dependent on variations in delays or data rates between the two modules. This method uses a packet of control information that is passed from the first module to the second module through the same path that passes the data between these modules.

9.2 The solution

A mini-packet-based scheme is disclosed for transferring control data together with graphics data between Window Descriptor Controller and Graphics Display Engine. Every window on the current display line has a corresponding mini packet structure comprising of a header followed by the data body residing in the Graphics FIFO. The two-word header describes the window as shown in Table 1 and 2.

TABLE 1. Mini packet header word #1 format.

32	31-28	27	26	25-24	23-16	15-0
data type	gfx type	first window	top/bottom	alpha type	window alpha	color

Each valid word in the Graphics FIFO has a header/data-body indicator by its MSB: 1 for header and 0 for data body.

TABLE 2. Mini packet header word #2 format.

32	31-28	25-16	10	9-0

data type	blank pix count	left edge	filter enable	window size

The fields in the first header word are defined as:

gfx type: window graphics type

first window: indicates this is the first window on a scan line

top/bottom: indicates this line is a top or bottom edge of the window

alpha type: alpha type of the window

window alpha: alpha value of the window as a whole

color: color of the window in rgb16 format, where appropriate

The fields in the second header word are interpreted as:

blank pix count: count of blank pixels at window's left side

left edge: start location of the window on a line

filter enable: YUV444 to YUV422 using filter

window size: actual window width excluding blank pixels

An empty line (no windows) is indicated by a packet with header word #1 only in which graphics type equals to 1111 binary.

A new line of graphics is indicated by the "first window" flag in word #1.

When the last window on the last line is done, the empty-line header is needed to put into the FIFO so that the Graphics Display Engine can release the line for display.

It is important to note that packetized data structures have been used primarily in the communication world where large amount of data needs to be transferred between hardware using a physical data link (wires for example).

The idea is not known to have been used in the graphics world where localized and small data control structure needs to be transferred between different design entities without requiring a off-chip large memory as a buffer. The header structure, and the way that the data/header is distinguished, the start/end of window, the end of active windows in a field as well as an empty line are handled, are all important in this embodiment. A general-purpose FIFO is all it needs and no other hardware is required. Routing is also simple because the write port of the FIFO is the only interface.

10 Color Look-up Table Loading Mechanism

10.1 The problem

For window surface based display, there are multiple windows on the same display screen with different graphics formats. For windows using a color look-up table (CLUT) format, it is often required to load specific color look-up table entries from external memory to on-chip ram before the window is displayed. This causes the following difficulties in design:

1. When to load the CLUT. For window formats RGB or YUV, there is no need to use the CLUT. Therefore no CLUT loading is required. For window format of CLUT, the CLUT needs be loaded before the window is displayed.

2. How many entries need be loaded to CLUT. There are different CLUT formats available, such as 2-bit CLUT (CLUT2), 4-bit CLUT (CLUT4), 8-bit CLUT (CLUT8). For CLUT2, a minimum of 4 entries need be loaded. For CLUT4, a minimum of 16 entries need be loaded. For CLUT8, all 256 entries need be loaded.

3. Where in the external memory to load from. The CLUT contents could reside in any location in the external memory. Different window contents may use CLUT from different memory locations.

In a preferred embodiment, the same window descriptor is used to control CLUT loading.

Each window on the display screen is defined with a window descriptor. The window descriptor defines the memory starting address of the graphics contents, the x position

on the display screen, the width of the window, the starting vertical display line and end vertical display line, window layer, etc. The same parameters could be used to define the CLUT loading: use graphics contents memory starting address to define CLUT memory starting address; use the width of graphics window to define the number of CLUT entries to be loaded; use the starting vertical display line and end vertical display line to define when to load the CLUT; use window layer to define the priority of CLUT loading if several windows are display in the same time (display line).

In this embodiment, two bits in the window descriptor to distinguish CLUT loading operation and normal display. They are defined as shown in TABLE 3, below:

TABLE 3. Window Operation Format

win_op	Function
00	Normal display
01	CLUT loading
10	Reserved
11	Last window descriptor

It is important to note the following aspects of the CLUT loading schemes of the present invention which:

1. Resolve the complication of synchronizing CLUT loading with window display. Even two windows with different CLUT tables can be displayed on the same display line.

2. Only the minimum required entries need to be loaded, instead of requiring all the entries to be loaded every time. This saves memory bandwidth and enables more functionality.

3. CLUT loading is very flexible and easy to control, which is suitable for various applications.

4. Simplify the hardware design. The same state machine for the window controller is used for CLUT loading. It also

shares the same DMA logic and layer/priority control logic as window controller.

5. Simplify software programming.

11 Graphics Line Buffer Control Scheme

11.1 The problem

The function of a Graphics Line Buffer is to temporarily store composed (blended) graphics images so that a graphics filter can be used to perform vertical filtering and scaling operations to generate output graphics images. The graphics composition logic (Graphics Display Engine) composites graphics images line by line using a clock rate that is significantly faster than the display pixel rate, and graphics filters run at the pixel display rate. The following problems occur:

1. How the buffer resources are controlled;
2. How the clock switching is done;
3. How the buffer is initialized;
4. How the buffers are cleared when they are filtered;
5. Both sides need to control the buffer, but multiple port solution is not practical.

11.2 The solution

In a preferred embodiment of the present invention,

1. Clock switching is done in the memory clock domain by the Graphics Display Engine using a vector of 7 bits, one for each line buffer in the set of line buffers (in the preferred embodiment there are 7). Glitch-free clock switching logic switches the clock according to this clock selection vector. The same clock switch vector is also used to mux the buffer control signals from the two sources.
2. Clock switching is done at field start and line start only avoiding the problem that the display (Graphics Filter) needs to access graphics data in real-time because at field and line starts, there is no active graphics data and clock switching can be done without causing glitches on the display end.
3. Clock switching requires dead cycle time and this information is indicated by a single 7-bit vector returned to the Graphics Display Engine to be

compared with its local expected pattern.

4. All buffers are cleared at every field start by the Graphics Display Engine to the equivalent of transparent black pixels so that graphics data can be blended the same way in any given time. Regardless of whether how many windows are composited into a line buffer, including zero windows, the Graphics Filter always gets the correct pixel data.

5. The release of a buffer by the Graphics Filter is indicated by an asynchronous signal to the memory clock domain at each line start. Once this signal is recognized, an internal buffer usage register is updated and then clock switching is performed to allow the Graphics Display Engine to work on the new buffer.

Important aspects of this buffer control schemes are:

1. Simple clock switching scheme;

2. The novel and unique way of passing buffer usage information between two clock domains;

3. Clever utilization of video inactive region for clock switching;

4. Unique way of implementing buffer clear;

5. Glitch-free of asynchronous buffer usage indicator passing from display clock to memory clock domain;

6. No need for RAM of multiple ports (> 2) and even a single-port RAM can be used.

7. When Graphics Filter accesses the composited graphics data on the graphics line buffer, it will clear the buffer(s) with black transparent pixels if it will not use the buffer(s) to generate the next graphics line.

12 Shared Video Scaling Engine

12.1 Problem

In certain applications of graphics and video display hardware, it may be necessary to scale the size of a motion video image either upwards (larger) or downwards (smaller),

but not scale it in both directions at the same time. It is also desirable to minimize the requirement for memory usage and memory bandwidth demands. Therefore it is desirable to scale down before writing to memory, and to scale up after reading from memory, rather than the other way around in either case. Conventionally there would be either separate hardware to scale down before writing to memory and to scale up after reading from memory, or else all scaling would be done in one location or the other, such as before writing to memory, even if the scaling direction is upwards.

In a preferred embodiment of the present invention, a single video scaling engine is configured such that it can be used for either scaling down the size of video images before writing the video images to memory, or for scaling up the size of video images after reading them from memory. In the former case the scaling engine is in the path between the video input and the memory write port. In the latter case the scaling engine is in the path between the memory read port and the display output. Therefore the scaling engine can be seen to exist in two distinct logical places in the design, while in fact occupying only one physical implementation.

This function is achieved by arranging a multiplexing function at the input of the scaling engine, with one input to the multiplexer being connected to the video input port and the other connected to the memory read port. The memory write port is arranged with a multiplexer at its input, with one input to the multiplexer connected to the output of the scaling engine and the other connected to the video input port. The display output port is arranged with a multiplexer at its input, with one connected to the output of the scaling engine and the other input connected to the output of the memory read port.

In the preferred embodiment, there are different clock domains associated with the video input and the display output functions of the chip. The video scaling engine uses a clock that is selected between the video input clock and the display output clock. The clock selection uses a glitch-free clock selection logic, i.e. a circuit that prevents the creation of extremely narrow clock pulses when the clock selection is changed. The read and write interfaces to memory both use asynchronous interfaces using FIFOs, so the memory clock domain can be distinct from both

the video input clock domain and the display output clock domain.

13 An All Digital Composite Video Decoder using Chroma Locked Sample Rate Converter with an Arbitrary Clock Frequency

Analog composite video has a chroma signal frequency interleaved in the luma signal. This chroma signal is modulated on to a sub-carrier frequency of 3.579545 MHz. The Luma signal covers a frequency span of zero to 4.2 MHz. The highest quality method of separating the luma from the chroma is to sample the video at a rate that is a multiple of the chroma sub-carrier frequency, and use a "comb" filter on the sampled data. This method imposes the limitation that the sampling frequency must be a multiple of the chroma sub-carrier frequency. Using such a chroma locked sampling frequency generally imposes significant costs and complications on the implementation, as it requires the creation of a sample clock of the correct frequency, which itself requires a stable, low noise controllable oscillator (e.g. a VCXO) in a control loop that locks the VCXO to the chroma burst frequency. In addition, such a chroma-locked clock frequency is often unrelated to the other frequencies in a large scale digital device, requiring multiple clock domains and asynchronous internal interfaces.

The disclosed solution is an IC that samples the analog video at a frequency that is essentially arbitrary and that is greater than four times the chroma sub-carrier frequency, while the sampling frequency does not need to be locked to the input video signal in phase or frequency. The sampled video data then goes through a sample rate converter (SRC) that down-converts the data to an effective rate four times the chroma sub-carrier frequency (4Fsc). This and all subsequent operations are performed in digital processing on a single integrated IC. The effective sample rate of 4Fsc does not require a clock frequency that is actually at 4Fsc, rather the clock frequency can be almost any higher frequency, such as 27MHz, and valid samples occur on some clock cycles while the overall rate of valid samples is equal to 4Fsc. The decimation (down-sampling) rate of the SRC is controlled by a chroma phase and frequency tracking module. The chroma phase and frequency tracking module looks at the output of the SRC during the color burst time interval and continuously adjusts the

decimation rate in order to align the color burst phase and frequency. In other words, the digital logic implements the logical equivalent of a phase locked loop (PLL), where the chroma burst phase and frequency are compared in a phase detector to the effective sample rate, which is intended to be 4Fsc, and the phase and frequency error terms are used to control the SRC decimation rate. The decimation function is applied to the incoming sampled video, and therefore the decimation function controls the chroma burst phase and frequency that is applied to the phase detector. This system is a closed feedback that functions in much the same way as a conventional PLL, and its operating parameters are readily designed in the same way as those of PLLs.

This solution places the least number of restrictions on the sampling clock. The sampling clock can run at the IC's system clock frequency or at the clock frequency of the destination of the decoded digital video. If the sampling clock is running at the system clock you will have a lower cost IC than one that has a system clock and a sub-carrier locked video decoder clock. A one clock IC will cause less noise or interference to the Analog to Digital Converter on the IC.

The system is all digital, and does not require an external crystal or Voltage Controlled Oscillator.

14 Time Base Corrector

We need the capability to mix or overlay graphics with decoded composite (analog video), which requires synchronizing the two image sources exactly. The conventional solutions are (1) use a composite video decoder that produces a line-locked clock that is phase-locked to the horizontal line rate of the composite video, and use that line-locked clock as the time base for the graphics and the display output, or (2) capture decoded composite video to a frame buffer, at display both the video from the frame buffer and the graphics from its source using a common display clock that need not be locked to the line rate of the incoming video. The former solution tends to introduce jitter on the display, since the phase locked loop (PLL) of the decoder has inherent timing noise and it also must track the timing of the incoming video, which is also variable. Also, this solution is not possible with some video decoder designs, as some of them do not operate on the basis of a line-locked clock. The latter

solution is relatively expensive in that it requires the memory space and memory bandwidth to capture decoded video in memory and to display it from memory; typically this requires two full frame buffers, and the bandwidth for both capture and display is twice the raw video data rate.

In the preferred embodiment, a line-based time base corrector is used. Composite video is decoded in whatever way is convenient, with whatever clock rate is convenient. For example, the decoder may sample video with a free-running clock operating at 27MHz, it may convert the sampled video via a sample rate converter to a sample rate that is four times the chroma subcarrier frequency, i.e. 14.318MHz, and it may produce output samples that are again converted via another sample rate converter to a sample rate of 858 times the horizontal line rate, or nominally 13.5MHz, with the electrical timing all using the same 27MHz clock used for input sampling. Note that the 27MHz clock, being free-running, is not locked to the line rate nor the to the chroma frequency of the incoming video. Decoded video samples are stored in a memory storage buffer of some substantial size, such as the length of one video scan line. Larger or smaller buffers may be used. The buffer is arranged to operate as a FIFO. Video samples are output from the FIFO at a clock rate that is nominally the correct rate, such as 13.5MHz, and this frequency is generally unrelated to the specific timing of the input video. The display timing is controlled by the input video timing in one regard only: the start of each display field follows the start of every input video field by a fixed delay, which delay is approximately equal to one-half the amount of time that would be required to fill the entire FIFO with video samples at the nominal video sample rate.

In the preferred embodiment, the video decoder operates as outlined above, with a 27MHz free running input sample rate, a first sample rate converter to convert the sampled video to a chroma-locked sample rate, chroma decoding and sync and other processing following this conversion, and a second sample rate converter to convert the video samples to a line-locked 13.5MHz sample rate. The electrical clock that controls the logic operates at the same frequency as the input sample clock; the sample rates of 14.318MHz and 13.5MHz are implemented by enabling valid data on only a subset of the clock cycles. The FIFO is the size of one line of active video at 13.5MHz, i.e. 720 samples by 16 bits per sample. The nominal delay amount of this FIFO is

therefore one video line period. At the start of every field of input video - this is the same as the end of the previous field - a sync signal is created that represents the vertical sync signal from the input video, however it is delayed from by one-half a video line period from the normal vertical sync time. The normal vertical sync time is aligned with either the horizontal sync signal or with a time that is half-way between successive horizontal sync signals, depending on whether the current video field is an even or off field. The generated sync signal immediately starts an output field of video and graphics, with the horizontal sync signal at the output synchronized with the generated sync signal. During the course of each video field, the display rate and the video rate entering the FIFO are in general different, due to differences between the actual frequencies of the display clock, which operates at a nominal 13.5MHz, and the effective input sample rate, which is also nominally 13.5MHz but which is also locked to operate at 858 times the horizontal line rate of the video input. Since the rates of data entering and leaving the FIFO are different, the FIFO will tend to either fill up or empty, depending on which rate is faster and which is slower. Using a FIFO size of one video line, and an initial delay at the start of every field of one-half a line time, the FIFO is approximately one-half full during the first active video line of every field.

As long as the accumulated change in FIFO fullness - in either direction - is less than one-half a video line, the FIFO will neither underflow nor overflow during the video field. This fact ensures correct operation when the display clock frequency is anywhere within a fairly broad range centered on the nominal frequency. Since the process is repeated every field, the FIFO fullness changes do not accumulate beyond one field time. The sync signals at the output of this time base corrector function are not exactly compliant with normal composite video (e.g. NTSC) standards, due to the instantaneous reset of the sync timing at the start of every field as described above.

However, this is not a problem with real-world television sets, as they are designed to accommodate adjustments of sync timing during the vertical blanking interval, when this adjustment occurs. Such sync timing adjustments are inherent in all normal VCRs, for different reasons, and of course TVs accommodate such timing. Since the display clock uses an arbitrary clock frequency that need only be

nominally equal to the ideal rate of e.g. 13.5MHz, the display clock can provided by an oscillator with no appreciable timing jitter, such as a 13.5 MHz crystal oscillator, or an oscillator operating at a multiple of 13.5MHz and divided down to this rate. Since the display clock has effectively no jitter, there is no jitter on the display, proving very quality graphics and video display, which would not be available from many conventional designs.

Without loss of generality, the FIFO in the time base corrector can be larger or smaller than one video line. The minimum size acceptable is determined by the maximum expected difference in the video source sample rate and the display sample rate. Larger FIFOs allow for greater variations in sample rate timing, however at greater expense. For any chosen FIFO size, the logic that generates the sync signal that initiates display video fields should incur a delay from the input video timing of one-half the delay of the entire FIFO, as described above.

Referring now to Fig. 5, a window controller block diagram is provided for window list sorting as used by some of the aspects of the invention described above.

1.1.1 Window Descriptor Definition

The information contained in window descriptor is used to control the reading of raw graphics data, conversion of this data to the common YUV-alpha format, blending of the various layers, position on the display screen and scaling.

win_operation[1:0]: window operation

- **0: normal display**
- **1: reload clut**
- **2: not used**
- **3: last window descriptor**

win_layer[3:0]: window layer, 0 to 15, with 0 to be the most bottom layer

win_xstart[9:0]: starting pixel of window display

win_ystart[9:0]: starting line of window display

Referring now to Fig. 6, a DMA block diagram is provided for use with the present invention.

1.2.1 Implementation

- DMA State Machine

```
IDLE          = 0   // idle state
NEW_LINE      = 1   // new line header write request
NEW_LINE_ACK  = 2   // new line header acknowledge
NEW_WIN       = 3   // first new win header write request
NEW_WIN1      = 4   // second new win header write request
NEW_WIN_ACK   = 5   // new win header acknowledge
GFX_REQ       = 6   // graphics memory data read request
GFX_ACK       = 7   // graphics memory data acknowledge
```

- FIFO flag generation

The empty, full, header room, data room flags are generated in this submodule. The graphics fifo has a depth of 32 levels. The read pointer and write pointer of the graphics fifo are 6-bit each which can count up to 64 levels. The empty flag is generated if all the bits of read pointer are equal to the write pointer. The full flag is generated if all the bits except the most signicant bit of read pointer are equal to write pointer. The header_room flag is generated when the fifo has room for at least 3 entries. The data_room flag is gen-

Referring now to Fig. 7, the video_sub component of the present invention is described.

Video_sub is the key component of BCM7014, processing all graphics and digital video data. It can process and composite one MPEG video source, one analog video source and up to 8 windows of graphics data of many formats simultaneously and then pass the composite YUV422 video to video encoder for direct TV output. The video inputs can be scaled up or down, captured and/or displayed. The graphics data for each window, stored in external SDRAM, are fetched and controled by the mem_block and dma_block in the CPU subsystem. The interface beetween these two subsystem is the FIFO (gfx_fifo). It store not only data but a 2 words header for each window, telling the graphics hardware how to process them. The HW will convert all the active windows's image line by line, to a common YUV422 format. All the active window's image are merged and stored in the display line buffer for post filting.Finally the graphics and 2 sources of video and background color are merged and streamed out to video encoder.

Referring now to Fig. 8, the time base corrector used in the present invention is shown and described with respect to a VDIN block diagram.

VDIN process two types of video sources: MPEG digital video and digitized analog video. MPEG digital video in CCIR656 8 bits format is decoded to extract the timing and data information. Digitized analog video are corrected in the time domain for jitter by the Time Based Corrector (TBC).

One of the processed source can be selected as pass-through video and will go to `composite` to blend with other sources for final output. One of the source can also be captured into external SDRAM. If `scale_down` is specified, it will go to the scaler first for scaling down to any specified factor. Data transfer are done by DMA mechanism. This block supply DMA address, byte_count and data to the dma and memory control block in CPU_sub to perform the data transfer.

Referring now to Fig. 9, a video scalar block according to the present invention is described.

The Scaler performs two-dimensional filtering on the incoming video, 4 taps vertically and 8 taps horizontally. Both down-scaling and up-scaling are supported.

In the case of down-scaling, the filter is enabled and incoming video is subsampled and subsequently stored to the external SDRAM. In the case of up-scaling, however, the filter is disabled and the incoming video is stored without change to the external SDRAM. The up-scaling is actually done when the video sample is taken out from the SDRAM as a video window. This scaler is shared between up-scaling and down-scaling filtering, so they can not operate at the same time.

With regard now to Fig. 10, the graphics display engine of the present invention is described.

This is part of the main graphics display pipeline in BroadCom Graphics Engine (BGE). It receives the graphics data and their format information in the form of a mini packet from the other major block, Window Descriptor Controller, through a graphics FIFO, converts them, blends them one on top of the other and finally writes to the local line buffers to be used by the downstream processing logic called Graphics Filter/Scaler.

The following tasks are performed by the Graphics Display Engine:

1. depacketize graphics packet data from graphics FIFO;
2. color conversion from native format to YUV including color lookup for CLUT images and RGB to YUV for rgb type of graphics;
3. perform YUV444 to YUV422 conversion, apply filtering if necessary;
4. pixel alpha derivation;
5. graphics layer blending;
6. manage line buffer timing and usage.

TABLE 2. Mini packet header word #2 format.

32	31-28	25-16	10	9-0
data type	blank pix count	left edge	filter enable	window size

The fields in the first header word are interpretated as:

gfx type:	window graphics type
first window:	first window indicator on a line
top/bottom:	top/bottom edges of the window
alpha type:	alpha type of the window
window alpha:	alpha value of the window as a whole
color:	color of the window in rgb16 format

The fields in the second header word are interpretated as:

blank pix count:	blank pixel count at window's left side
left edge:	start location of the window on a line
filter enable:	YUV444 to YUV422 using filter
window size:	actual window size excluding blank pixels

An empty line (no windows) is indicated by a packet with header word #1 only in which graphics type equals to 1111 binary.

A new line of graphics is indicated by the "first window" flag in word #1.

When the last window on the last line is done, the empty-line header is needed to put into the FIFO so that the Graphics Display Engine can release the line for display.

Graphics Display Engine Operation: The Graphics Display Engine determines its state by checking the availability of the seven line buffers. Following is a typical sequence that it will go through:

1. After system reset, all line buffers are indicated to be available;
2. When a field sync is detected, Graphics Display Engine will first clear all line buffers with equivalent "black pixels" (black YUV422 pixels with its alpha equal to zero). At the mean time, Window Descriptor Controller will start to put data into Graphics FIFO;
3. After line buffers are cleared, data is processed whenever the Graphics FIFO is not empty, line by line until no graphics line buffer is available;

With regard now to Fig. 11, a state diagram of the state machine of the present invention is disclosed.

1. LD_RESET;
2. GFX_HEADER1;
3. CLK_SWITCH;
4. GFX_HEADER2;
5. GFX_CONTENT;
6. PIPE_COMP.

With regard now to Fig. 12, the data flow path of a graphics converter according to the present invention is disclosed.

Fig 12 shows the data flow of YUV data and Alpha generation for different data format. All data come from mini packet and go to different branch depend on its data format. Both yuv_data and alpha output come with different delays to be used later in the 444to422 (win_filt) stage. Header I,II decodeing and line buffer address generation logic (ld_waddr, ld_raddr) also shown un the diagram.

With regard now to Fig. 13, the data flow path for conversion and blending in the present invention is shown.

Fig. 13

shows the dataflow of 444to422 (win_filt) operation and the final blending stage. Alpha, U, V are filtered (1-2-1 filtering) seperately. The three branches in U,V filtering are : filter_on, filter_off and yuv422 source which do not require filtering.

ld_rdata is the selected line buffer to be blended. For luma: both multiplications are done in gblend module; For U,V: one of the multiplication is done early in the process. The output of luma,chroma and alpha merged into blend_data and then written back to line buffer.

With regard now to Fig. 14, the structure of the graphics color look up table (CLUT) is disclosed.

Graphics Color Look-up Table (Graphics CLUT) allows CLUT image to get its color and alpha value by indexing to a RAM-based table.

With regard now to Fig. 15, the write port timing is shown in graphical form.

Graphics CLUT Controller and CLUT RAM. The RAM is a 256x32 dual-port SRAM corresponding to the 256 entries in the CLUT.

The write port is controlled by Window Descriptor Controller using the following signaling:

clut_mem_req encompasses clut_mem_wr such that the rising edge is used to clear the internal write pointer. The succeeding clut_mem_wr increments the write pointer from 0 upto 255.

With regard now to Fig. 16, a graphics line buffer according to the present invention is disclsoed.

Graphics Line Buffer is a bank of seven line buffers. Four line buffers are used by graphics anti-flicker/SRC filter to generate current graphics display line and the other three are updated/filled to hold three more graphics scan lines as spare. The maximum number of graphics lines to be updated is 3.

The Graphics Line Buffer allows dynamic clock switching between two clock sources. The clock selection can be from a third source.

buffer controller.

Inside the graphics line buffer controller are seven clock muxes and muxes selecting ram control and data signals for each of the line buffers. ld_csel_mem is used to select clock source for the synchronous line buffers. A clock selection pattern (clk_enb_vect) is returned back to the Graphics Dispaly Engine to indicate the end of clock switching.

With regard now to Fig. 17, a block diagram of the digital video decoder according to the present invention is described.

With regard now to Fig. 18, a table is provided for the port definition of the present invention.

3.0 Port Definitions

Table 2: VDEC top level interface Definition

Signal	Bit Range	IO	To/From	Description
clk_27mhz		IN	<- EXT	27 MHz vdec clock
clk_27mhz_not		IN	<- EXT	27 MHz inverted vdec clock
vdec_reset_		IN	<- EXT	external reset
ad_i	[9:0]	IN	<- EXT	10 bit video data from ADC.
vdec_nabts_st	[8:0]	OUT	-> EXT	NABTS status bits
vdec_cc_st	[17:0]	OUT	-> EXT	Status bits and CC data
data_out	[15:0]	OUT	-> EXT	vdec Y C output
dvalid		OUT	-> EXT	dvalid signal for *data_out*.
tbc_vsync		OUT	-> EXT	VSync signal to TBC
tbc_hsync		OUT	-> EXT	HSync signal to TBC
tbc_field		OUT	-> EXT	Field type signal to TBC.
clamp		OUT	-> EXT	clamp control signal
nabts_rclk		IN	<- EXT	Read clock for NABTS data.
nabts_raddr	[6:0]	IN	<- EXT	Read address bus for NABTS data
nabts_rdata	[31:0]	OUT	-> EXT	NABTS data bus .
vdec_test	[15:0]	OUT	-> EXT	Test bits for VDEC section.
vdec_coef_wr		IN	<- EXT	
vdec_fe_csz_t		IN	<- EXT	

Fig. 1: Complete graphics and video display pipeline

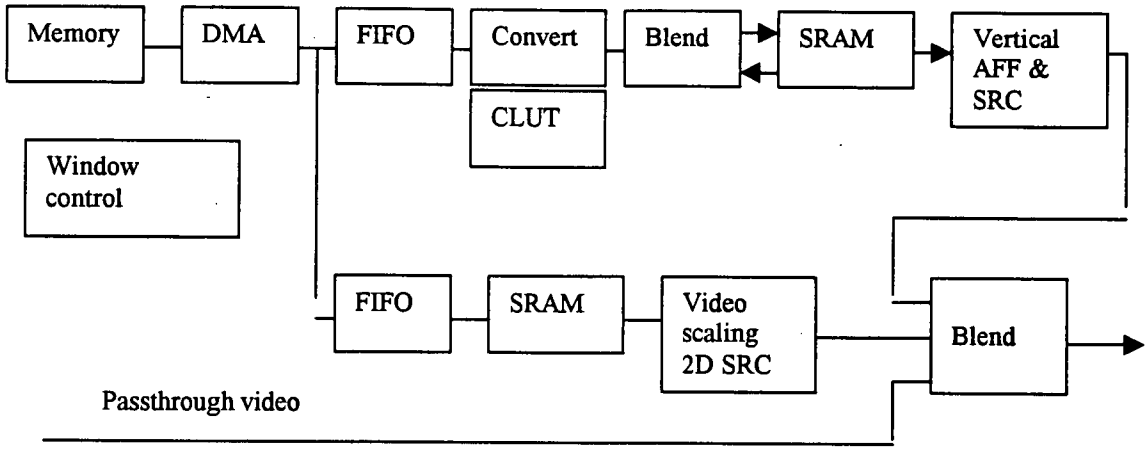


Figure 2: Graphics display pipeline

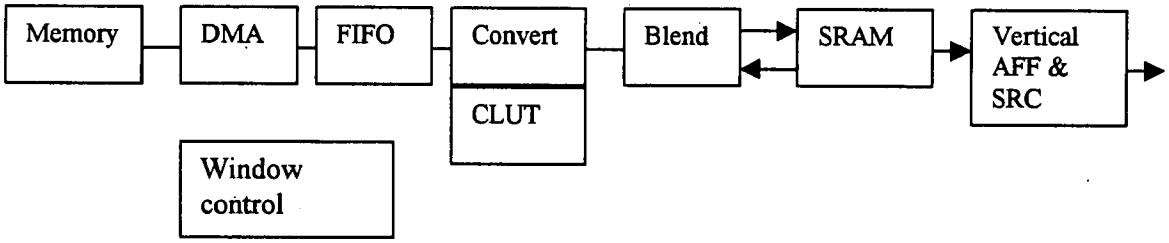


Fig. 3: Four-tap Gaussian filter kernel

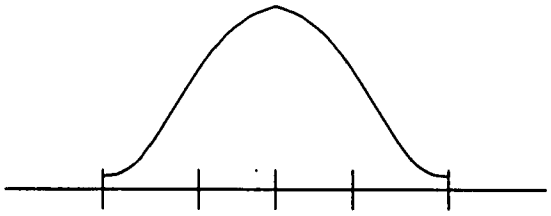


Fig. 4: Two-tap bilinear filter kernel

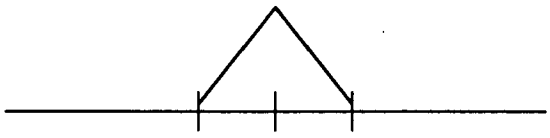


FIG. 5

Window Controller Block Diagram

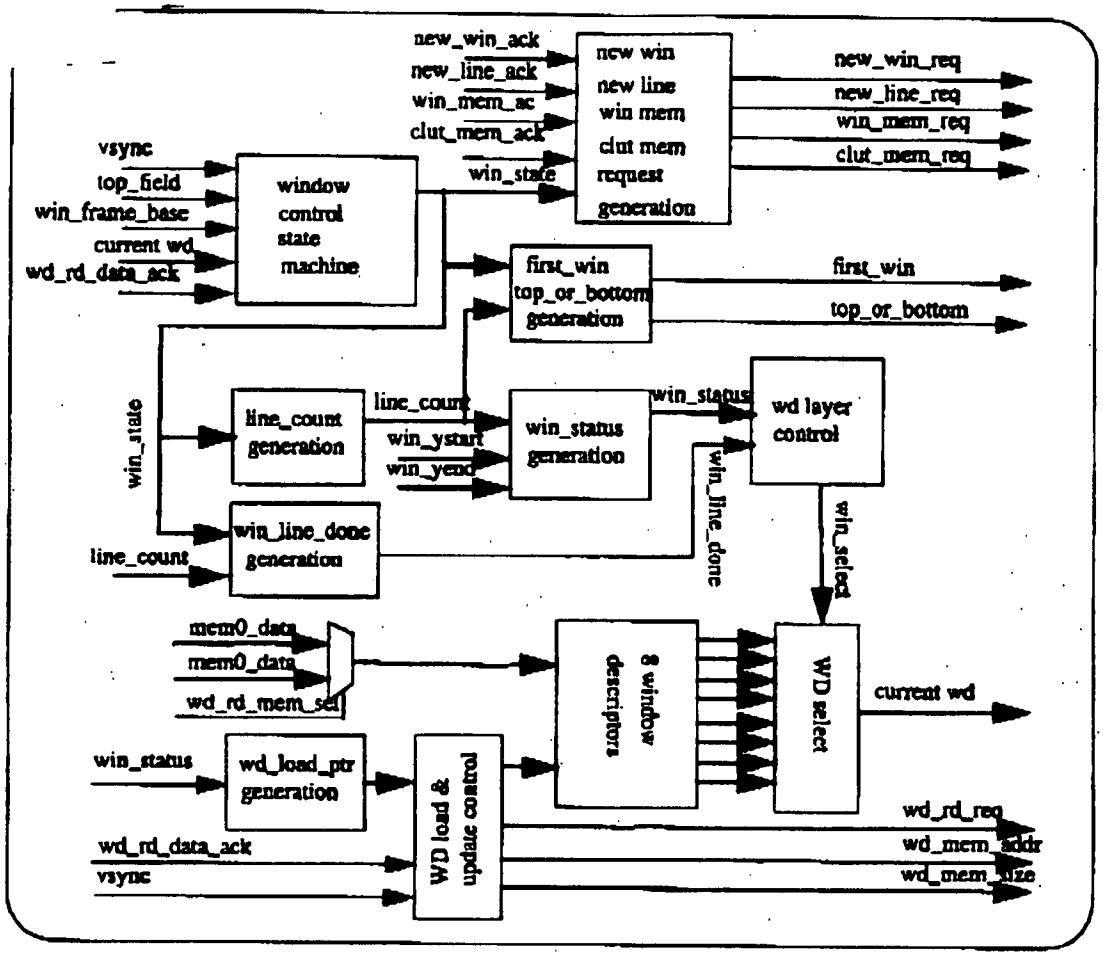


Fig 6

DMA Block Diagram

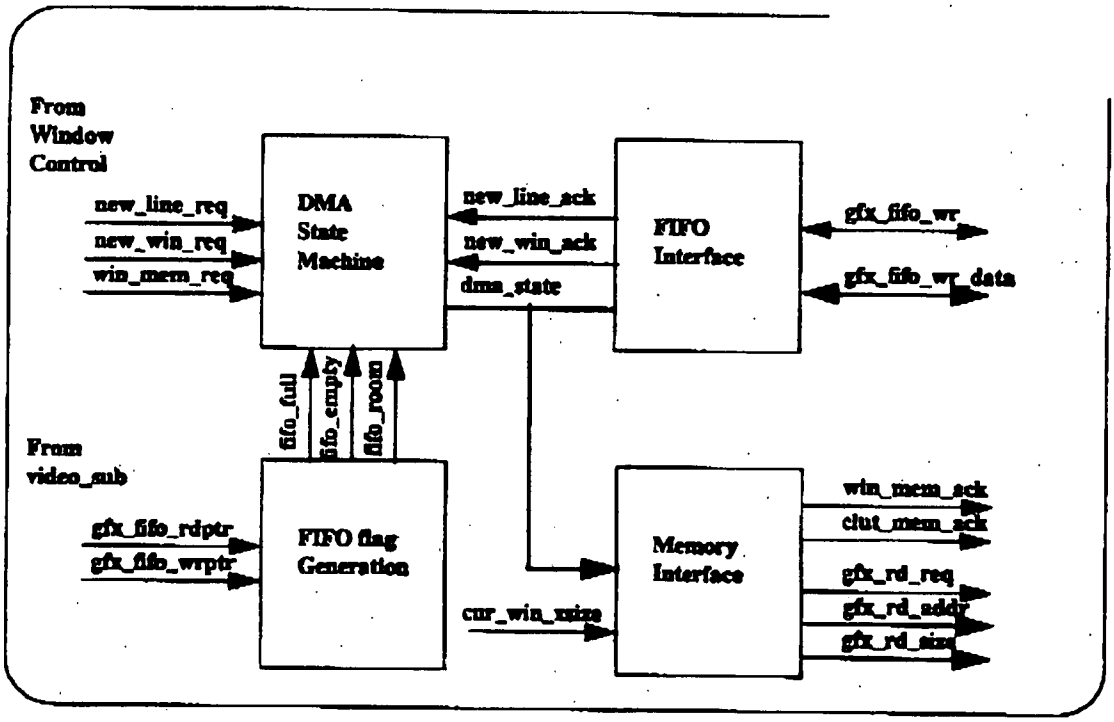


FIG. 7

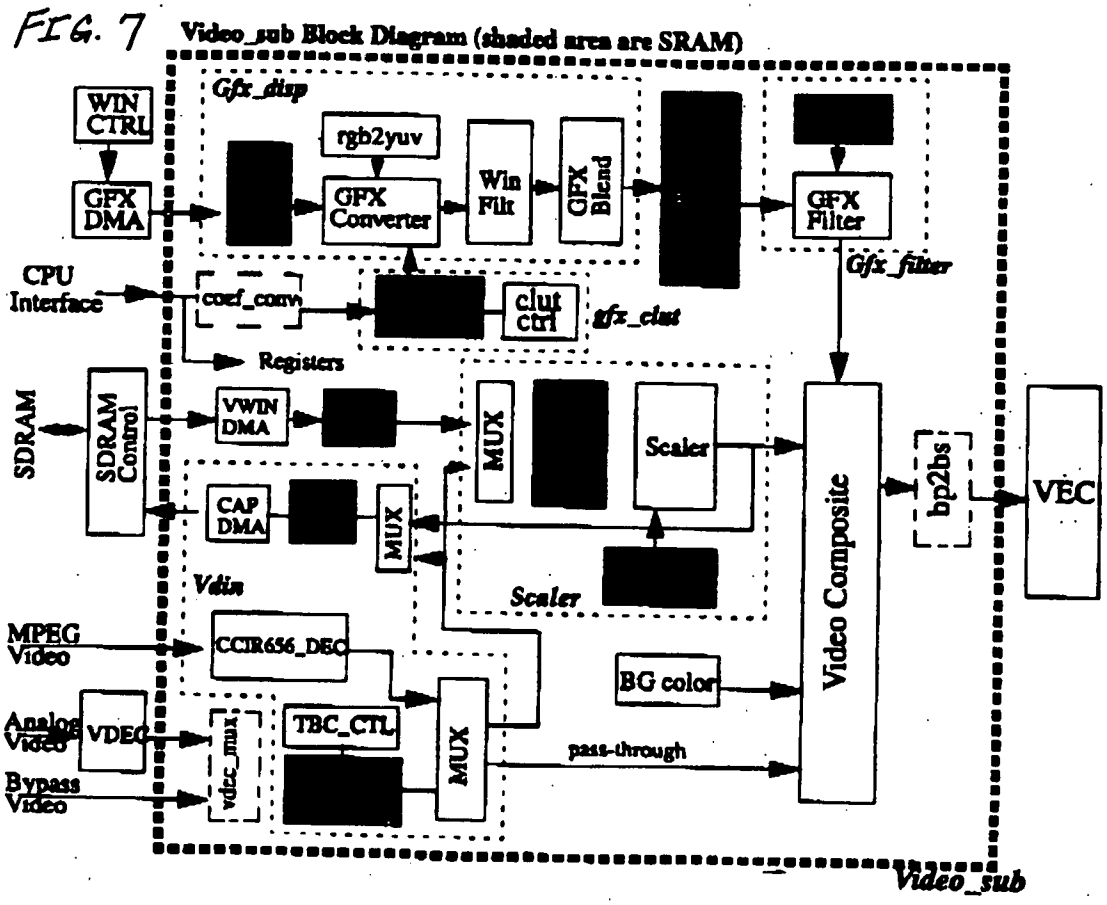

Video_sub Block Diagram (shaded area are SRAM)
Gfx_disp
WIN CTRL
GFX DMA
rgb2yuv
GFX Converter
Win Filt
GFX Blend
GFX Filter
Gfx_filter
CPU Interface
coef_conv
clut ctrl
gfx_clut
Registers
SDRAM
SDRAM Control
VWIN DMA
MUX
Scaler
CAP DMA
MUX
Vdin
Scaler
MPEG Video
CCIR656_DEC
BG color
Video Composite
bp2bs
VEC
Analog Video
VDEC
vdec_mux
TBC_CTL
MUX
pass-through
Bypass Video
Video_sub

Fig 8

VDIN Block Diagram

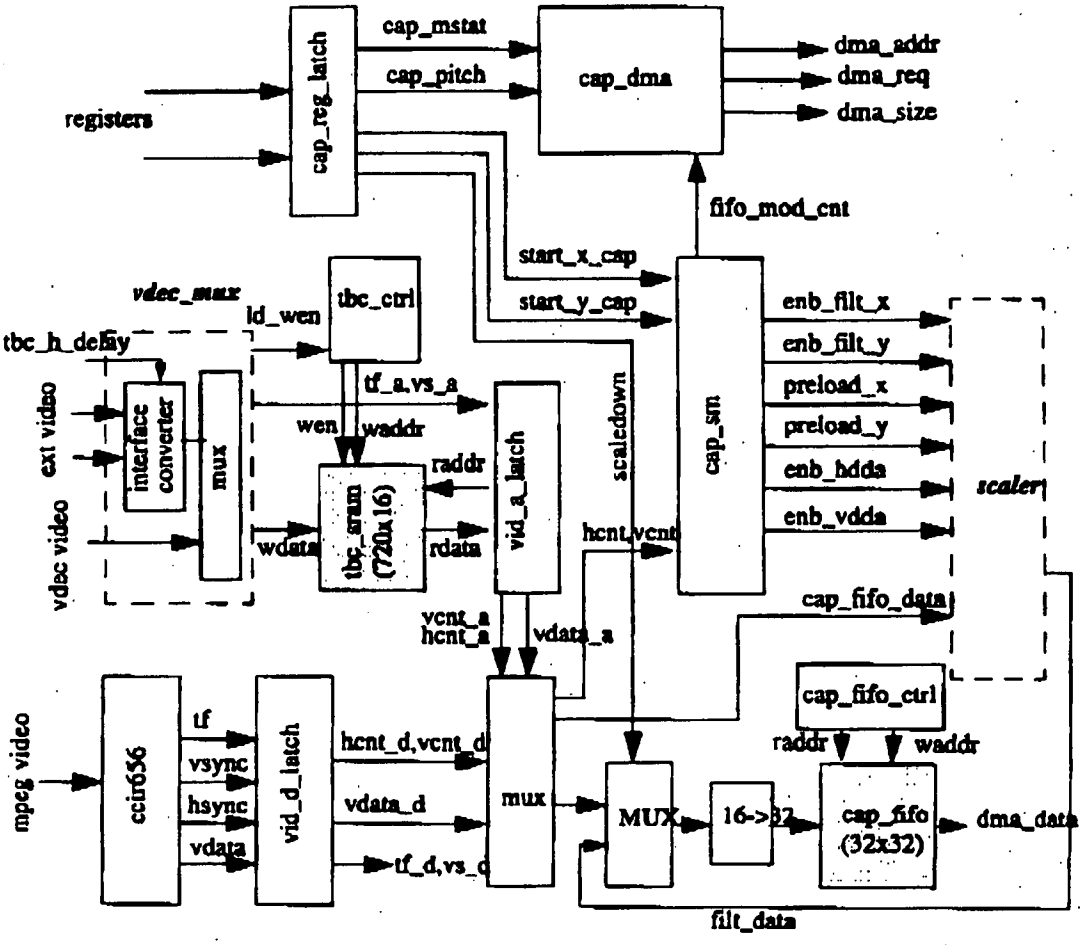
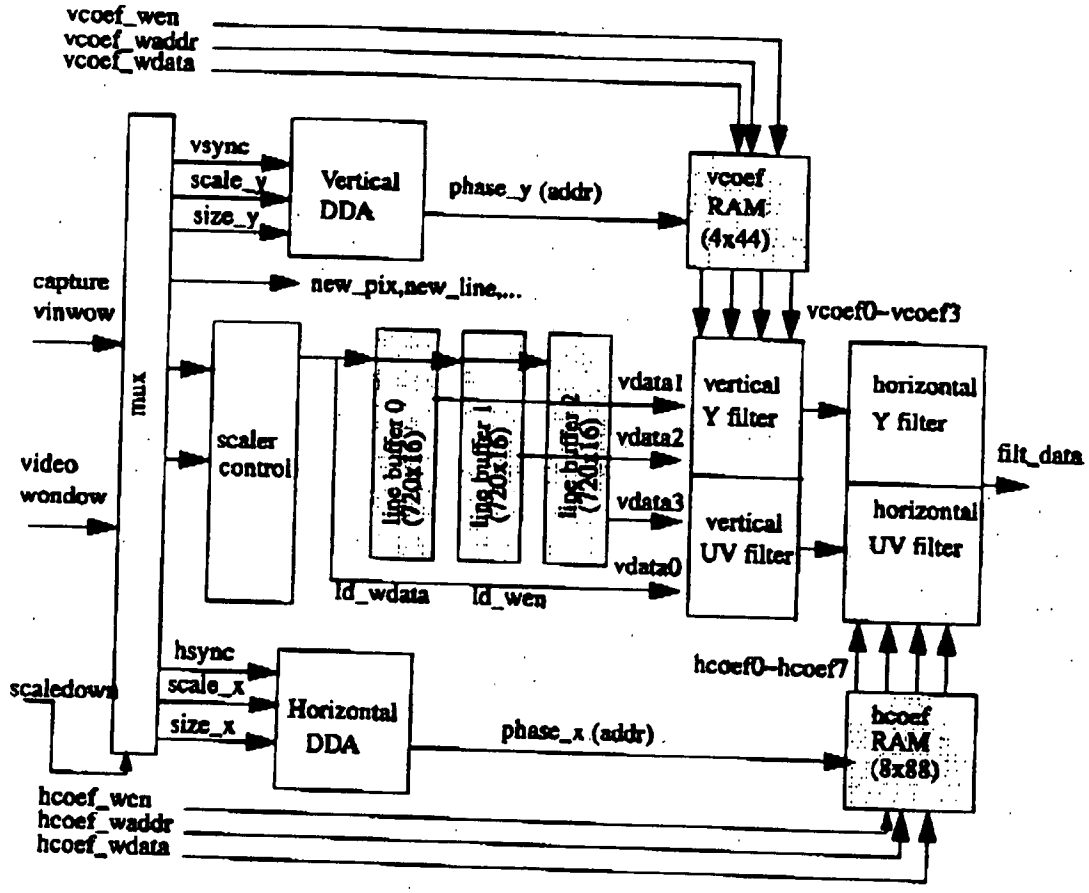
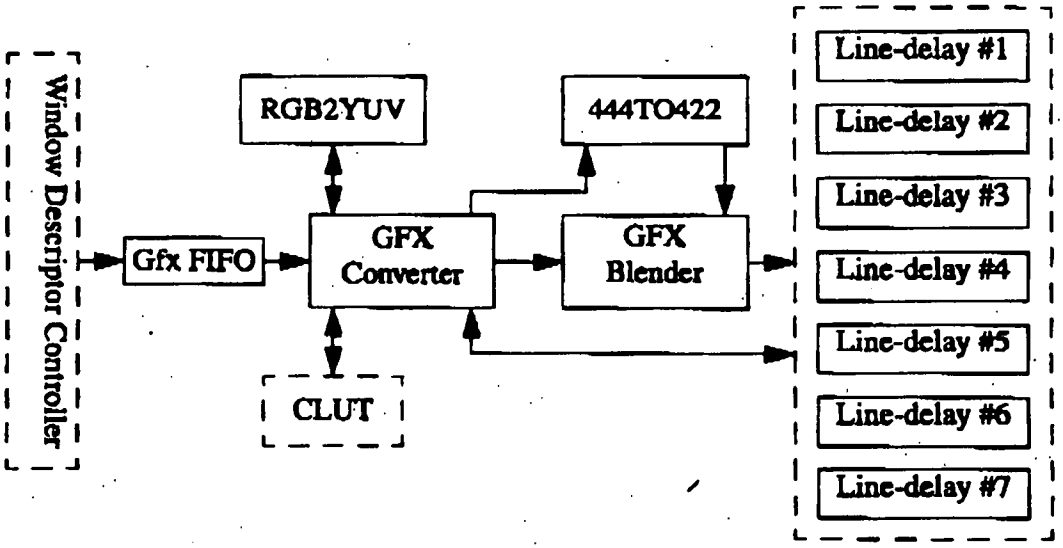


Fig 9

Video Scaler Block Diagram



As shown in Fig 10 Graphics Display Engine is composed of five main components:



Graphics Display Engine.

Following is a state diagram of this state machine:

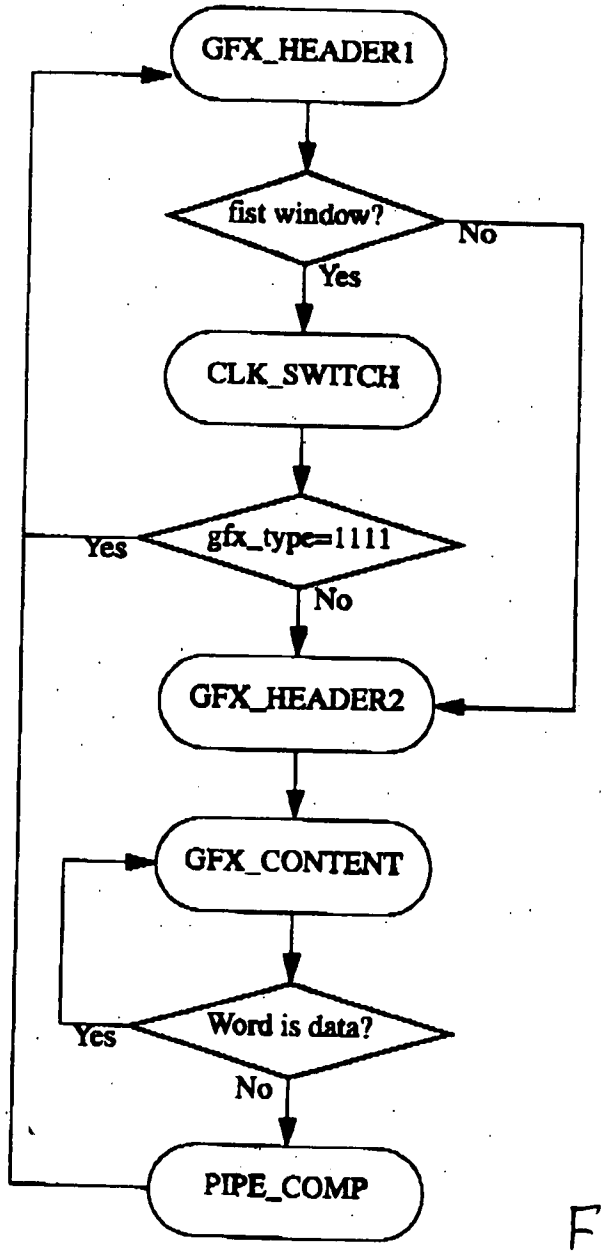
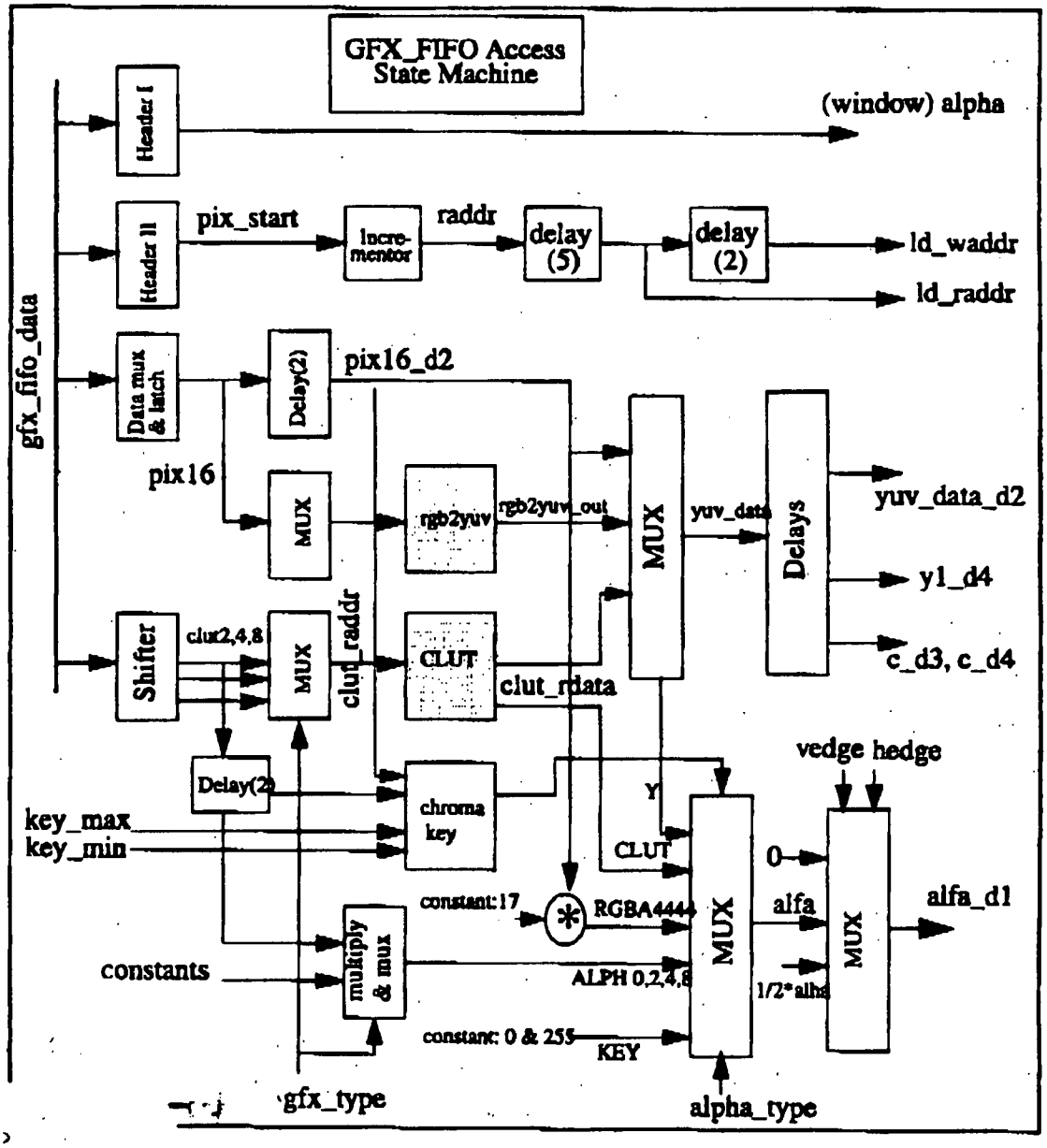


Fig 11.



GFX_Converter DataPath

Fig 12

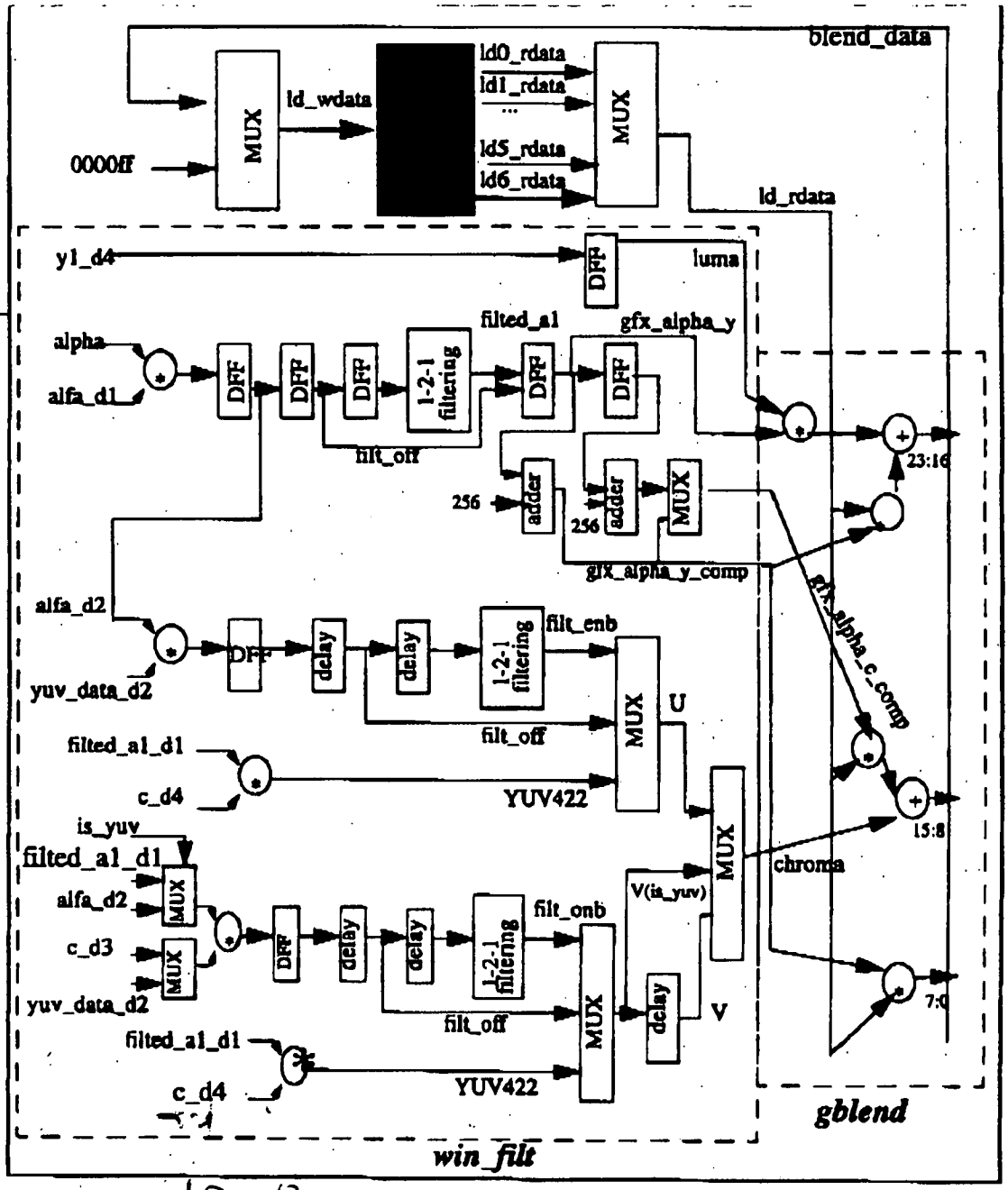


Fig 13

. Win_Filt 444to422 Conversion and gblend DataPath

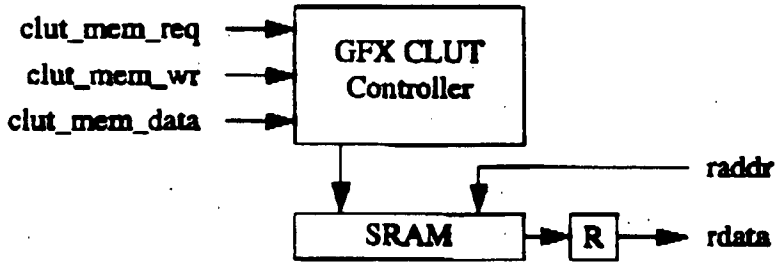


Fig 14

Graphics CLUT block diagram.

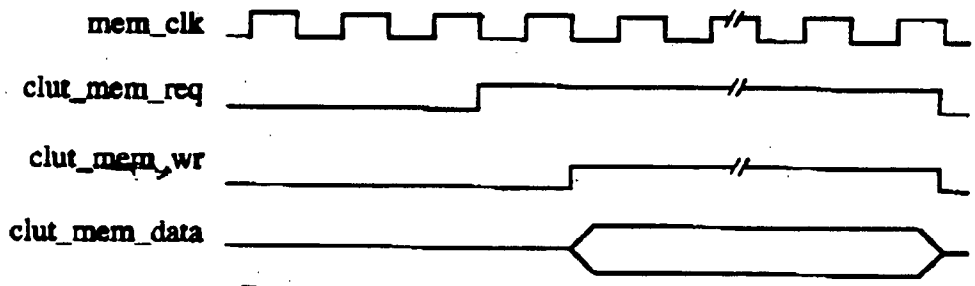


Fig 15

Graphics CLUT write port timing.

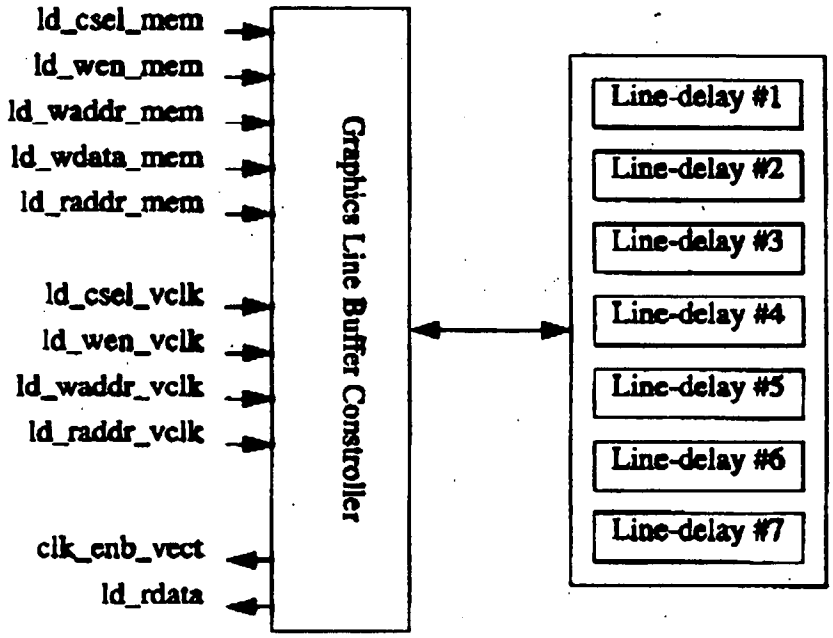


FIGURE Graphics Display Engine.

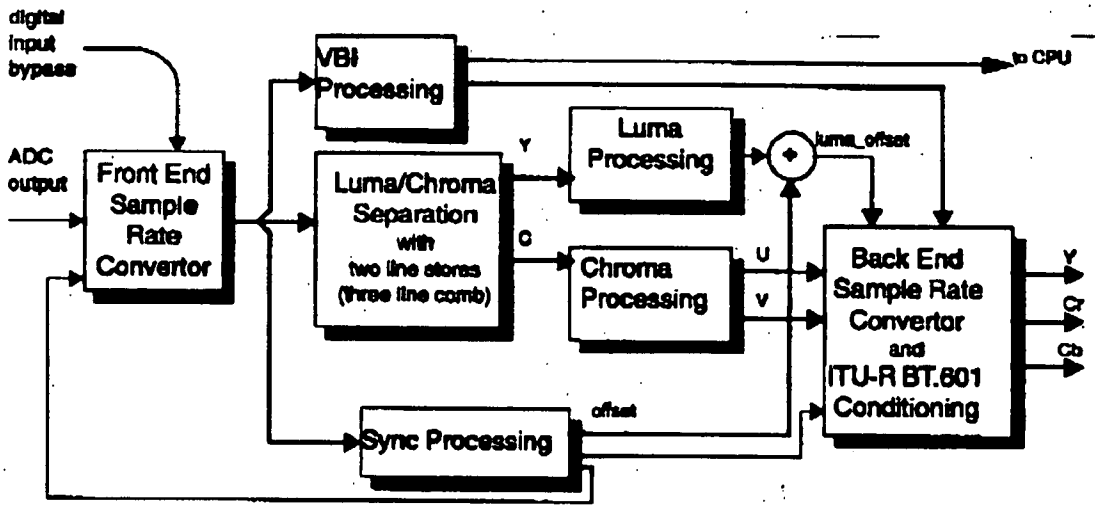

digital input bypass
ADC output
Front End Sample Rate Convertor
VBI Processing
to CPU
Luma/Chroma Separation with two line stores (three line comb)
Y
C
Luma Processing
luma_offset
Chroma Processing
U
V
Back End Sample Rate Convertor and ITU-R BT.601 Conditioning
Y
Cr
Cb
Sync Processing
offset

Fig 17.

FIG. 18

3.0 Port Definitions

Table 2: VDEC top level interface Definition

Signal	Bit Range	IO	To/From	Description
clk_27mhz		IN	<- EXT	27 MHz vdec clock
clk_27mhz_not		IN	<- EXT	27 MHz inverted vdec clock
vdec_reset_		IN	<- EXT	external reset
ad_i	[9:0]	IN	<- EXT	10 bit video data from ADC.
vdec_nabts_st	[8:0]	OUT	-> EXT	NABTS status bits
vdec_cc_st	[17:0]	OUT	-> EXT	Status bits and CC data
data_out	[15:0]	OUT	-> EXT	vdec Y C output
dvalid		OUT	-> EXT	dvalid signal for *data_out*.
tbc_vsync		OUT	-> EXT	VSync signal to TBC
tbc_hsync		OUT	-> EXT	HSync signal to TBC
tbc_field		OUT	-> EXT	Field type signal to TBC.
clamp		OUT	-> EXT	clamp control signal
nabts_rclk		IN	<- EXT	Read clock for NABTS data.
nabts_raddr	[6:0]	IN	<- EXT	Read address bus for NABTS data
nabts_rdata	[31:0]	OUT	-> EXT	NABTS data bus .
vdec_test	[15:0]	OUT	-> EXT	Test bits for VDEC section.
vdec_coef_wr		IN	<- EXT	
vdec_fe_csz_t		IN	<- EXT	

SERIAL NUMBER	FILING DATE	CLASS	GROUP ART UNIT	ATTORNEY DOCKET NO.
60/107,875 PROVISIONAL	11/09/98		0000	

APPLICANT

ALEXANDER G. MACINNIS, SAN CARLOS, CA; CHUNGFUH TANG, SARATOGA, CA; XIAODONG DON XIE, SANTA CLARA, CA; JAMES PATTERSON, SANTA CLARA, CA; GREG KRANAWETTER, SAN JOSE, CA.

CONTINUING DOMESTIC DATA********
VERIFIED

_____

371 (NAT'L STAGE) DATA********
VERIFIED

_____

FOREIGN APPLICATIONS********
VERIFIED

_____

FOREIGN FILING LICENSE GRANTED 12/08/98

Foreign Priority claimed 35 USC 119 (a-d) conditions met ☐yes ☐no ☐yes ☐no ☐Met after Allowance Verified and Acknowledged _____ Examiner's Initials _____ Initials	STATE OR COUNTRY	SHEETS DRAWING	TOTAL CLAIMS	INDEPENDENT CLAIMS
	CA	15		

ADDRESS

IRELL & MANELLA
1800 AVENUE OF THE STARS
SUITE 900
LOS ANGELES CA 90067

TITLE

GRAPHICS CHIP ARCHITECTURE

FILING FEE RECEIVED	FEES: Authority has been given in Paper No. _____ to charge/credit DEPOSIT ACCOUNT NO. _____ for the following:	☐ All Fees ☐ 1.16 Fees (Filing) ☐ 1.17 Fees (Processing Ext. of time) ☐ 1.18 Fees (Issue) ☐ Other _____ ☐ Credit
$150		